conference

proceedings

3rd USENIX Windows NT Symposium

Seattle, Washington, USA
July 12–15, 1999

Sponsored by
The USENIX Association

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
WWW URL: http://www.usenix.org

The price is $18 for members and $24 for nonmembers.
Outside the U.S.A. and Canada, please add
$10 per copy for postage (via air printed matter).

Past Windows NT Proceedings

2nd USENIX Windows NT Symposium	1998	Seattle, Washington, USA	$18/24
USENIX Windows NT Workshop	1997	Seattle, Washington, USA	$18/24

ISBN 1-880446-29-4

Printed in the United States of America on 50% recycled paper, 10-15% post consumer waste.

USENIX Association

Proceedings of the

3rd USENIX Windows NT Symposium

July 12–15, 1999
Seattle, Washington, USA

Symposium Organizers

Symposium Co-Chairs

Werner Vogels, *Cornell University*
Stephen Walli, *Softway Systems, Inc.*

Symposium Steering Committee

Michael B. Jones, *Microsoft Research, Microsoft Corp.*
Andrew Hume, *Bell Laboratories*
Thorsten von Eicken, *Cornell University*

Program Committee

Brian Bershad, *University of Washington*
Gary Campbell, *Compaq Computer Corporation*
Andrew Chien, *University of California, San Diego*
Thorsten von Eicken, *Cornell University*
Jim Gray, *Microsoft Research, Microsoft Corp.*
Michael B. Jones, *Microsoft Research, Microsoft Corp.*
Sam Leffler, *VMware, Inc.*
Richard Oehler, *IBM T.J. Watson Research Center*
Susan Owicki, *InterTrust Technologies Corporation*
Karin Petersen, *Xerox Palo Alto Research Center*
David Steere, *Oregon Graduate Institute*
Ramu Sunkara, *Oracle Corporation*
Rumi Zahir, *Intel Corporation*
Myron Zimmerman, *VenturCom, Inc.*

Advanced Workshop Committee

Todd Needham, *Microsoft Research, Microsoft Corp.*
Werner Vogels, *Cornell University*

The USENIX Association Staff

Contents

3rd USENIX Windows NT Symposium

July 12–15, 1999
Seattle, Washington, USA

Monday, July 12

Cluster Computing
Session Chair: Werner Vogels, Cornell University

Porting
Session Chair: Stephen Walli, Softway Systems, Inc.

High-Performance Systems
Session Chair: Jim Gray, Microsoft Research, Microsoft Corporation

Tuesday, July 13

Real Time and Not
Session Chair: Susan Owicki, InterTrust Technologies Corporation

Indirection
Session Chair: Michael B. Jones, Microsoft Research, Microsoft Corporation

Internet
Session Chair: Karin Petersen, Xerox Palo Alto Research Center

Efficient User-Level Thread Migration and Checkpointing on Windows NT Clusters[†]

Hazim Abdel-Shafi, Evan Speight, and John K. Bennett
Department of Electrical and Computer Engineering
Rice University
Houston, Texas
{shafi, espeight, jkb}@rice.edu

Abstract

Clusters of industry-standard multiprocessors are emerging as a competitive alternative for large-scale parallel computing. However, these systems have several disadvantages over large-scale multi-processors, including complex thread scheduling and increased susceptibility to failure. This paper describes the design and implementation of two user-level mechanisms in the Brazos parallel programming environment that address these issues on clusters of multiprocessors running Windows NT: thread migration and checkpointing. These mechanisms offer several benefits: (1) The ability to tolerate the failure of multiple computing nodes with minimal runtime overhead and short recovery time. (2) The ability to add and remove computing nodes while applications continue to run, simplifying scheduled maintenance operations and facilitating load balancing. (3) The ability to tolerate power failures by performing a checkpoint before shutdown or by migrating computation threads to other stable nodes. Brazos is a distributed system that supports both shared memory and message passing parallel programming paradigms on networks of Intel x86-based multiprocessors running Windows NT. The performance of thread migration in Brazos is an order of magnitude faster than previously reported Windows NT implementations, and is competitive with implementations on other operating systems. The checkpoint facility exhibits low runtime overhead and fast recovery time.

1. Introduction

Recent advances in multiprocessor and network performance have made cluster-based computing an increasingly cost-effective option for large-scale parallel computing. Distributed systems constructed of industry-standard multiprocessors and networks offer an excellent price-to-performance ratio compared with monolithic multiprocessor systems, especially for large systems. Advances in user-level communication mechanisms [4], memory consistency models [12, 14], and network technologies (e.g. [4] and [9]) have improved the performance of these systems significantly. However, several issues limit the widespread adoption of clustered multiprocessors for distributed parallel computing. First, clusters of multiprocessors have an increased risk of failure simply because there are more components that might fail. This tendency to fail is exacerbated by the presence of other users running applications that might cause the system to crash. Second, multiple instances of the operating system running concurrently on each node require more administration and maintenance than a single operating system. Finally, reducing inequitable load distributions in parallel applications may require a system-level solution. Addressing these issues necessitates effective thread migration and checkpointing capabilities.

In this paper we describe the design and implementation of user-level mechanisms for thread migration and checkpointing. Together, these mechanisms support the following functionality:

- Tolerating the failure of multiple computing nodes with minimal runtime overhead and recovery time.
- Addition and removal of computing nodes while applications continue to execute.
- Handling power failures on systems with limited power backup by performing a checkpoint before shutdowns are triggered, or by migrating computational threads to other stable nodes.
- Distribution of the computational load among nodes in a cluster.

We have implemented both mechanisms within the Brazos system [21], a Windows NT-based parallel programming environment that runs on industry standard networks of Intel x86-based multiprocessors. In addition to providing programmers with the

[†] This research was supported in part by grants and donations from Compaq Computer Corporation, Intel Corporation, Microsoft Corporation, Packet Engines, Inc., and by the Texas Advanced Technology Program under Grant No. 003604-022.

abstraction of running on a single shared memory multiprocessor, Brazos supports message passing by implementing the MPI library [20].

Thread migration in the context of a distributed system involves the movement of a computation thread from one currently executing process to another running process. Thread migration has been previously proposed as a tool for load-balancing and communication reduction in distributed shared memory systems [13, 23]. Our work extends the use of thread migration to fault tolerance and cluster management. Migration can be used to tolerate shutdowns due to scheduled maintenance or power loss by dynamically moving all computation threads and necessary data of the application to another available node, without restarting the application. Migration can also be used to add or remove multiprocessor nodes on-the-fly by relocating existing computation threads to the new nodes as appropriate. Finally, the runtime system or programmer may elect to migrate a thread to another node in cases where moving the thread to the data is a better option than moving the data to the thread.

Applications that run for a long time or that require high-availability need a means of recovering from failures, while minimizing the runtime overhead required to ensure recoverability. Previous work in distributed fault tolerance schemes can be categorized as either transaction or checkpoint-based, although combinations of both have been used. Transaction-based recovery is similar to database recovery, in that the distributed system maintains a list of memory transactions or messages [5]. Single node failures can be tolerated by replaying the transactions related to the failed node. Checkpointing is used to save the state of a process. In case of a failure, the checkpoint files are applied and computation can proceed from the point of the last checkpoint [1, 22]. Systems that combine transactions and checkpoints attempt to minimize the amount of work lost due to failure as well as the space requirements for recovery data.

Our implementation of checkpointing is distinguished in two ways. First, we minimize the amount of data saved during a checkpoint operation by leveraging some of the existing coherence-related information available in the Brazos runtime system. This reduces both the overhead required to create checkpoints and the time needed to recover from failures. Second, our checkpoint facility can be initiated either explicitly upon user request or implicitly using predetermined checkpointing intervals. Our results indicate that the facility, given an appropriate choice of checkpoint interval, exhibits low execution time overhead and fast recovery times.

The rest of the paper is organized as follows. In Section 2 we described the design and performance of the Brazos thread migration mechanism. Section 3 contains a similar analysis of the Brazos checkpointing mechanism. In Section 4, we describe how thread migration and checkpoints can be combined to perform several fault tolerance and cluster management functions. Related work is discussed in Section 5. We conclude and describe future research directions in Section 6.

2. Thread Migration

This section describes the design issues that must be addressed to implement a thread migration mechanism, as well as the specific implementation decisions appropriate for use in a Windows NT environment.

2.1. Thread Contexts and Stacks

A thread in Windows NT is comprised of the processor's register set (or context), a thread-specific stack, and a Windows NT-specific area called Thread Local Storage (TLS) [18] intended to contain data instanced per thread. In order to migrate a thread from one process to another, a thread's stack, context, and the user portions of the TLS (Brazos uses the TLS for certain runtime system information) have to be packaged and sent to the remote node. Upon receiving the thread migration message, the remote process copies the contents of the remote thread's stack into a local thread's stack and *injects* the context and TLS data of the remote thread into that of the local thread.

Since stacks may include pointers that reference stack data, a mechanism must be in place to guarantee that these pointers have the same meaning on the new host (issues related to heap and shared memory coherence are addressed in Section 2.3). Furthermore, the stack contains the saved state from any functions executed before migration, implying that both nodes must have the program code loaded at the same virtual address. Code location is not an issue for Brazos, since a distributed application executing on the Brazos system employs multiple instances of the same program.

Two solutions addressing stack data pointer management have been proposed [13, 23]. First, the destination host can scan the received stack data and adjust or remap any pointers encountered by adding the appropriate offset. This solution has several potential problems:

- Stack data may be misaligned, making it difficult to identify the location of pointers.

- Actual variables stored on the stack may contain values that are similar to stack addresses. Changing such values will likely result in incorrect computation.
- Stack pointer values may reside in processor registers, making it necessary to also examine register contents and adjust them accordingly.

These problems make the scanning or remapping of pointers unattractive in the general case. We have chosen to adopt the alternative solution, in which the destination thread's stack must be located at the same virtual address as the source thread's stack [13, 23]. We ensure that the stacks of both threads begin at the same virtual address by reserving the thread stack space for all user threads that may exist during the execution of the distributed process. During the Brazos runtime system initialization on each node, we create a number of threads equal to the total number of user threads executing on all nodes. A potential problem with this approach is the memory and operating system overhead for each thread created. Upon thread creation, Windows NT reserves a default 1MB region from the process' virtual address space for the thread's stack. However, only 2 pages (a total of 8KB on x86-based systems) of memory are initially committed. The amount of memory committed for a stack then increases as needed. The operating system also reserves the necessary internal data structures needed for manipulating and scheduling threads. Since user processes may address up to 2GB of virtual memory (3GB on Enterprise Server systems), we believe that the cost in terms of the memory space wasted by idle user threads is relatively low. In addition, since typical thread stack sizes are often much less than the default 1MB provided for by Windows NT, the amount of wasted address space may be reduced by lowering the maximum thread stack size to a more appropriate value.

2.2. Win32 Support for Thread Migration

Because Windows NT threads are managed by the operating system, we need a mechanism to find a thread's stack and context before we can perform migration. The Win32 API provides several functions that are used to manipulate both thread state and virtual memory [18]. A thread's context may be acquired and set using the GetThreadContext and SetThreadContext functions, respectively. In order to find the thread's stack, we first acquire the thread's current stack pointer, which is part of a thread's context. The Win32 VirtualQuery function is then used to determine the region of committed memory associated with the thread's stack. At this point, the thread's state is completely known, and the context and stack are copied into a message buffer. The migrating thread on the local machine is suspended using the Win32 SuspendThread call[1], and the migration message is sent to the destination node. Upon receipt of the message, the destination node sets the local thread's context, copies the stack data, and activates the thread using the ResumeThread routine. Since Brazos uses UDP, the destination node explicitly acknowledges that the migration message was successfully received.

Migrating threads with open files are handled as follows. The Win32 calls that access files are intercepted by a wrapper function that saves the parameters necessary to reopen the files after migration (i.e., the name of the file, its sharing mode, read/write mode etc.) [11]. In addition, the current file pointer values are determined using standard Win32 calls. This information is then transmitted to the new node and used to reopen the appropriate files and reset their file pointers. Since the file handles used by the migrated thread will be different than the handles created at the new node, we include a mechanism for mapping file handles from the handle used by the thread to the actual handle at the new node. The same wrapper functions used to save access parameters also take care of this mapping. This mechanism requires that all nodes have access to a common file system. File contents are not migrated; however, users have the option of flushing output file buffers prior to migration. A similar mechanism is used for checkpoint and recovery, as described in Section 3.3.

2.3. Ensuring Correctness

There are two correctness issues that arise when threads are allowed to migrate: the effects of thread migration on shared memory coherence mechanisms, and the management of static (linker allocated) or non-shared heap data during thread migration [13, 23]. As described earlier, a thread's context and its automatic or stack variables are maintained throughout the migration process; however, no explicit actions are performed with respect to shared or static memory that may be accessed by a thread. In order to move such data, we need a method to identify static variables or non-shared heap memory that are only accessed by the thread being migrated. Although it is possible to construct a memory map for a Windows NT process [22] and to migrate any such data detected, it is both difficult and time consuming to do so. In order to achieve fast

[1] We initially used Win32 synchronization primitives (e.g., WaitForSingleObject) to suspend and resume threads. During development, we discovered that in order to correctly set the thread's context, it must be executing in user code where it is suspended explicitly using SuspendThread.

thread migration, our approach requires that the programmer ensure that no read/write static or non-shared heap data is used as thread-private data. All read/write data private to a thread must be allocated in Brazos shared memory. Since the runtime system will automatically migrate private data allocated in shared memory when it is accessed by a thread, this rule ensures that private data can be accessed by the migrating thread on any node. Aligning such structures on page boundaries reduces the chances of other threads causing more communication through false sharing.

Shared memory coherence issues increase in complexity if relaxed consistency models are employed [23]. For example, Brazos implements a multiple writer protocol and two relaxed memory consistency models: Release Consistency (RC) [8] and Scope Consistency (ScC) [12]. Both of these models allow multiple nodes to modify different portions of a virtual page concurrently, and only perform coherence or consistency actions at specific synchronization points.

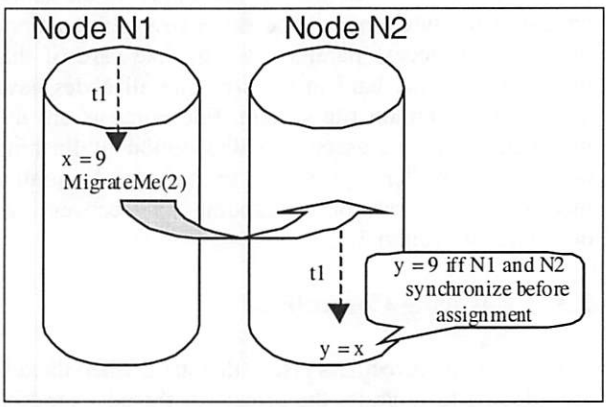


Figure 1: Correctness Issues in Thread Migration

Figure 1 identifies the problems that can occur during thread migration. Consider a thread *t1*, which initially resides on node *N1* and modifies a shared variable *x*. If thread *t1* is subsequently migrated to node *N2*, it may access the variable *x*, which could either contain a valid or stale value depending upon whether an appropriate synchronization event[2] occurred between the initial modification and the access following migration. The simplest approach to this problem requires that a global synchronization operation, such as a barrier, be performed before migration can take place. Another solution includes shipping coherence information with thread migration messages, possibly resulting in slower overall migration performance. Our current approach is to allow thread migration to occur whenever most appropriate, and to require the programmer or runtime system to synchronize as necessary prior to migrating.

2.4. Thread Migration API

Our implementation of thread migration adds a single function to the Brazos API: MigrateMe(DestNode). A thread calls this function when it wants to migrate itself to another node. When a thread invokes this function, it is suspended, a special thread called the *Migration Agent* awakens, and the calling thread is migrated to the specified node. At the destination node, the corresponding thread continues execution from the same point in the program. The thread on the original node remains indefinitely suspended unless the same thread migrates back.

2.5. Performance and Analysis

The performance measurements described here were performed on a network of Compaq Professional Workstation 8000's. Each system contains four 200MHz Pentium Pro processors with 256KB of L2 cache. The systems are equipped with two 33MHz 32-bit PCI network interfaces: Fast Ethernet and Gigabit Ethernet. Each node contains 256MB of main memory and an Ultra-Wide SCSI-3 disk controller and drive.

Parameter	FastEthernet	Gigabit Ethernet
Win32 calls	0.089 ms	0.089ms
Network/copy/synch.	1.121 ms	0.921 ms
Total Migration time	1.21 ms	1.01 ms

Table 1: Performance of Thread Migration in Brazos with 4KB Stacks

We measured the performance of thread migration by executing 1000 back-to-back thread migrations between two nodes. Each iteration contains two calls to the MigrateMe() function. Each call suspends the local thread and places the stack and context in a message destined for another node. At the receiver, an acknowledgement message is sent back to the originating node for reliability purposes[3]. The stack contents are copied, the context is injected into the local thread, and the local thread is then resumed. We also measured the overhead of the various Win32 calls used in the migration process by averaging over 1000 instances of each call. As shown in Table 1, thread migration takes between 1.01 and 1.21ms, depending on the network interface used. Communication accounts for the majority of the migration time.

[2] A barrier is sufficient for both RC and ScC. Locks are sufficient for RC, but for ScC the modification of *x* occurs within a lock's scope on *N1*. The same lock has to be acquired by a thread on node *N2* before the correct value of *x* may be read.

[3] There were no retries or dropped messages observed during the performance measurements presented here.

Win32 Function (No. of calls)	Cost per call
GetThreadContext (1)	27 μsec
SetThreadContext (1)	27 μsec
SuspendThread (1)	14 μsec
ResumeThread (1)	5 μsec
VirtualQuery (2)	8 μsec

Table 2: The Cost of Win32 Calls Used in Thread Migration

Table 2 identifies the fixed costs associated with migration. Certain costs associated with migration, such as the time needed to wake up the migration agent thread and the time needed to schedule threads after being migrated, are difficult to quantify. The overhead of the Win32 calls that we measured was low compared to the network, stack copying, and synchronization overheads.

The performance of the Brazos thread migration mechanism is an order of magnitude faster than that reported for the Millipede Windows NT-based system [13], and is competitive with other user-level thread migration implementations (see Section 5.1).

3. Checkpoint and Recovery

The ability to tolerate faults becomes increasingly important as the number of multiprocessors in a cluster grows. This is because the probability of failure increasing along with the complexity of the system: a system with sixteen power supplies is more likely to experience power supply failure than a system with one. This is an especially important concern for long-running applications. Additionally, clusters of multiprocessors may be geographically distributed with different local loads that vary over time. Clustered systems need to be able to adapt to these variations. In the extreme, it may be necessary to move all threads off a particular node, and then resume them elsewhere. Checkpoint and restart is an effective mechanism for this situation. Finally, since maintenance and upgrade functions typically require the interruption of service, fault tolerance support allows processors or systems to be shutdown and restarted, thus interrupting rather than terminating the running application. In the remainder of this section we describe the Brazos checkpointing mechanism.

3.1. Checkpointing in Brazos

In order to checkpoint a running Windows NT process, it is necessary to save its state. A process' state includes the stacks and contexts of all threads that exist in that process, the contents of memory (the heap and static data areas, code may not be important), and any operating system objects owned by the process, such as open file handles, synchronization objects, etc. In a networked computing environment, it is also necessary to recreate any network connections upon recovery. This may require additional information concerning the overall composition of the cluster to be saved. The amount of time needed to create a checkpoint is a performance concern because of the potential for a large amount of process-specific information. For a checkpointing mechanism to be practical, it must incur low overhead during normal operation and allow recovery in substantially less time than the potential time lost due to failure.

The structure of a Windows NT process within the Brazos environment can be broken down into the following components: user and runtime system threads, shared memory, static or other heap memory, operating system handles for various synchronization objects and files, and network connections to other nodes participating in the computation. We will discuss each of these components in the context of checkpointing and recovery. Before discussing these details, it is necessary to review the mechanics of starting a Brazos distributed process.

Before a user can execute a Brazos parallel application, a configuration file must be created. The configuration file contains information about the executable program, the names of nodes participating in the computation, and the number of user threads on each node. The program executable, configuration files, and input data files must exist on a shared disk volume accessible from all nodes. Every node that hosts a Brazos process runs a Windows NT service responsible for starting Brazos processes on the local node. The first node listed in the configuration file starts execution by sequentially sending process start requests to other nodes. To avoid deadlock, each Brazos process listens on two network ports: one for requests and the other for replies. The runtime system creates Brazos system threads to handle requests and replies from the network. Once the Brazos runtime system is initialized, the user's application program is started. Programs typically allocate shared memory and initialize data structures using input files at the beginning of execution. The application then proceeds until completion. Several characteristics of the Brazos runtime system facilitate checkpointing:

- The initialization of Brazos is identical for any given configuration file; therefore, it is not necessary to checkpoint most of the runtime system-specific entities. For example, it is not necessary to checkpoint the runtime system's threads, sockets, and most other data structures

because they can be easily recreated at recovery by rerunning the initialization routines.

- Brazos synchronization objects such as locks and barriers do not need to be saved because checkpoints are created only at barriers. Thus synchronization objects can be reinitialized during recovery.
- Since checkpoints are independent of the Brazos runtime system initialization process, it is possible to recover processes using different configuration files. This proves to be a valuable feature, allowing recovery on a different number of nodes than were present when the checkpoint is made. This allows a Brazos distributed application to be moved to accommodate varying system loads without seriously affecting performance.

3.2. Memory Issues

Although the Brazos parallel programming environment supports both shared memory and message passing programs, we only discuss issues related to shared memory checkpointing in this paper. The checkpoint facility must save a consistent view of memory to permit full recovery. Memory coherence in Brazos is maintained at the granularity of a virtual page. Operating system support for virtual memory is used to protect pages, and special virtual memory exception handlers are used by the runtime system to ensure that coherence is maintained. Since Brazos supports multiple writer protocols [3], if a node performs a store to an address on a page, a *twin* or duplicate copy of the page is made before the store is allowed to complete. The twin is later used to create a *diff* when another node requires a copy of the page. A diff is a list of addresses and values for locations that have been modified on a particular page. Since multiple nodes may modify a single page simultaneously, multiple nodes may contain diffs for each page. When a node faults on such a page, it will receive diffs from all nodes with modifications to the page in order to reconstruct a valid version of that page. The Brazos runtime system maintains the state of every shared memory page, and this state is used by the checkpoint facility to examine the status of every page and save the necessary coherence-related information, including diffs or twins.

3.3. Implementing Checkpoint

In order to guarantee that the state of pages is consistent and that no coherence actions are pending, we only create checkpoints at global synchronization points such as barriers. At the checkpoint, all data necessary to perform a recovery operation is stored. These data include all shared memory pages, including any twins and diffs and their related runtime system status data structures. In addition, all user threads and their contexts and stacks are saved using the methodology described in Section 2. Since Brazos includes multiple instances of barrier data structures, it is necessary to save the number of the barrier instance used at the time of creating the checkpoint. This ensures that the barrier structures are updated before threads are resumed after recovery. The time consumed in creating checkpoints is a function of the amount of shared memory in use at each of the nodes in the system, but is considerably smaller than the size of the running process (see Section 3.5).

Before a checkpoint is initiated at a barrier, all computation threads have to arrive at that barrier. The thread currently responsible for managing synchronization performs the necessary communication with other nodes to supply coherence information. When all nodes have arrived at the barrier, a message is sent to release all nodes. If a checkpoint is to be performed, threads are not immediately resumed upon receiving the barrier completion message. Instead, a *Checkpoint Agent* thread initiates the checkpoint. Application threads resume once the checkpoint is completed.

Open files during checkpoint and recovery are handled in much the same manner as open files during thread migration (see Section 2.2). File access functions are wrapped and the necessary parameters are saved. This information is stored in the checkpoint file and is used to reopen the files and to set their file pointers during recovery. File contents are not included in the checkpoint; however, users have the option of flushing output file buffers during checkpoint creation.

Two optimizations were added to our checkpointing facility. First, checkpoint files are initially written to each node's local disk to improve performance. In order to allow recovery from other nodes, we also copy the checkpoint files to a network file system. In order to hide the latency of the copy process, the checkpoint agent wakes up the computation threads as soon as the local checkpoint files are written. The copy is performed in the background while the application continues execution. Second, to minimize the size of checkpoint files, we modified the runtime system to keep track of shared memory pages that are allocated and used before the checkpoint takes place and only save information related to modified pages. This substantially reduces the size of checkpoint files by eliminating the need to copy the full pool of shared memory pages.

3.4. Programmer Interface

There are a small number of checkpoint-related issues that must to be addressed by the programmer. Since the checkpoint facility only saves shared memory pages, the programmer allocates all application data (except for stack variables) in shared memory, even if they will not be shared in practice. This restriction may be ignored only for static or heap data that is read-only (note that this restriction is already enforced in order to provide thread migration as described in Section 2.3). Variables that are declared and initialized in this manner will be reinitialized upon recovery and execution will proceed correctly.

Either the user or the runtime system may initiate a checkpoint. Programmers are able to specify a flag to a barrier call that instructs the system to perform a checkpoint or recovery operation. Since users will likely be more concerned with the amount of time potentially lost due to failure, Brazos also supports an automatic checkpoint interval that can be specified in the configuration file, on the command line, or at any point in time during execution using the Brazos user interface. Using the interval method, when the time since the previous checkpoint exceeds the specified interval, the runtime system instructs all nodes to perform a checkpoint. A user interface initiated checkpoint is performed at the next barrier instance. This feature is useful for planned shutdown situations and avoids checkpoint overhead unless necessary. All Brazos programs must explicitly include at least one recovery barrier. For programs that do not use barriers, our current implementation requires inserting additional synchronization. We are investigating techniques to perform checkpoint and recovery without barriers.

Brazos provides two types of recovery mechanisms: automatic recovery on the remaining nodes, or recovery with the addition of a new node to replace the failed node. For recovery on a replacement node, a new configuration file identifying this node is created using the Brazos user interface. Then, the application program calls the recovery process at a barrier placed after the runtime system and application initialization phases in each Brazos process. The only application initialization required before recovery is shared memory allocation and global read-only variable initialization. After recovery, all threads exit the recovery barrier and continue execution from the last checkpoint. During automatic recovery, the Brazos runtime system detects that a node has failed through the use of *heartbeat* messages that are sent out by a designated node every 10 seconds (the heartbeat period is user-selectable). When a node fails, the remaining nodes will restart their Brazos processes from the last successful checkpoint, and the threads from the failed node will be distributed to the surviving nodes in a round-robin fashion. We currently do not attempt to optimize the assignment of threads to nodes.

3.5. Checkpoint Performance

To demonstrate the checkpointing facility, we used two applications: Water, a molecular dynamics application from the SPLASH benchmark suite [19] using a 4096 molecule dataset, and a locally-written SOR (successive over relaxation) using 4000×4000 matrices. Both applications were run on four nodes, each with a single user thread. Water is intended to be representative of applications with a small shared memory footprint, whereas SOR is intended to be representative of applications with a large shared memory footprint. To measure the overhead of the checkpoint facility, we used the system initiated checkpoint mode while varying the checkpoint interval from two to thirty two minutes.

Ckpt interval in min (number of ckpts)	Execution time in minutes	Ckpt overhead as % of exec time
(none)	42.23	0%
32 (1)	42.40	0.39%
16 (3)	42.38	0.34%
8 (6)	42.58	0.82%
4 (11)	43.20	2.28%
2 (20)	43.88	3.90%

Table 3: Checkpoint Overhead vs. Interval Length for Water

Table 3 shows that for Water the overhead of performing checkpoints to network disks every two minutes was about 3.9% of the total execution time. The size of the Water process was about 45MB and the average checkpoint size was 5.2MB per node. Checkpoint time averaged about 1.25 seconds, and recovery time was 21 seconds (17 seconds for runtime system and user initialization, and 4 seconds for actual recovery) on four nodes. When only local files were created, the runtime overhead was essentially the same.

Ckpt interval in min (number of ckpts)	Execution time in minutes	Ckpt overhead as % of exec time
(none)	36.06	0%
32 (1)	36.45	1.08%
16 (2)	37.24	3.27%
8 (4)	37.97	5.30%
4 (9)	40.62	12.65%
2 (19)	46.18	28.06%

Table 4: Checkpoint Overhead vs. Interval Length for SOR

As Table 4 shows, the checkpoint execution time overhead of SOR (with a 122MB footprint) is relatively high for low checkpoint interval settings (28% for two-minute intervals). These results suggest that checkpoint intervals of eight minutes or higher are more appropriate in this case, resulting in runtime overheads of about 5% or less. The process size for SOR was about 163MB; the average size of the checkpoint file was 62.1MB. The average checkpoint time was 25.8 seconds, and recovery time was 46 seconds (19 seconds for runtime system and user initialization and 27 seconds for data recovery).

In order to improve checkpoint performance for large applications, we are adding support for incremental checkpoints. Incremental checkpoints reduce runtime overhead in two ways. First, they eliminate the need to stall the computation threads at barriers by allowing checkpoint operations to be performed in parallel with computation. Second, the amount of data copied to checkpoint files is reduced because only modifications made since the previous full checkpoint need to be saved. This is accomplished by tracking pages that have been modified since the last checkpoint and only saving the information necessary to apply these changes from the last checkpoint instance. We are presently implementing this mechanism.

4. Integrating Thread Migration and Checkpoint/Recovery

Thread migration can be used to allow the addition or removal of nodes participating in a distributed application. To add a node, the application is started on the new node and network connections are established with all nodes currently participating in the computation (as described by the configuration file) during the runtime system's initialization phase. The root process then takes note of the added node and sets a special flag. The current barrier manager uses this information to synchronize all existing nodes with the new node. Any threads that need to be migrated to the new node are moved before execution resumes.

Shared memory accesses on the new node are handled in the usual way by the Brazos runtime system; however, all static or non-shared heap data have to be initialized before the node can participate in the computation. For this purpose, an initial user thread is used to perform any required user data initialization. In addition, it is necessary to modify some runtime system data structures to reflect the addition of the new node. This work is accomplished at the barrier at which the new node is detected.

Removing nodes is basically the reverse process. An important difference is that once the computation threads have migrated to other nodes, the contents of shared memory at the node to be removed need to be transferred to other nodes before the process is terminated. This is accomplished by creating a checkpoint of the shared memory state of the node to be removed. All remaining nodes then apply the necessary changes to their state using the contents of the checkpoint file, and update the appropriate runtime system information to reflect the departure of a node before proceeding with computation.

In case of a power failure, our system triggers checkpoints of all running distributed processes before the system is shutdown. This requires that the system have an UPS with sufficient backup time to allow checkpointing to take place. If other nodes are not affected by the power failure, threads on the affected machine(s) migrate to other nodes using a methodology similar to that of removing and adding a node to the system.

5. Related Work

This section discusses previous relevant work in fault tolerance and thread migration on software distributed shared memory systems. Since we are not aware of any previous work that combines both techniques, related work is separated into two parts. First, we discuss other systems that utilize thread migration. Then we discuss various fault tolerance techniques proposed for distributed shared memory systems. Finally, we discuss work related to checkpointing Windows NT processes.

5.1. Thread Migration

Millipede [13] and D-CVM [23] both employ thread migration. Millipede is a Windows NT-based DSM system that includes a user-level thread migration mechanism similar to ours. In particular, they use the same method of ensuring that all thread stacks are at identical virtual addresses across all nodes, and impose the same restrictions on memory use. Millipede uses thread migration to reduce communication costs by keeping track of all accesses made by a thread that result in inter-node communication. Millipede requires that memory be sequentially consistent [17], which results in lower overall performance. The reported migration time on Millipede is 70ms on Pentium-based systems with 100Mbps Ethernet.

D-CVM uses thread migration to dynamically redistribute computation threads to nodes to reduce

communication and improve load-balancing [23]. Instead of relying solely on the tracking of page faults as in Millipede, D-CVM contains an active thread tracking mechanism. This mechanism tracks sharing among local threads by both serializing thread execution and adding per-page access counters for each thread in a DSM process. D-CVM's approach to coherence is similar to ours. It uses a multiple-writer LRC (Lazy Release Consistency [14]) coherence protocol and avoids correctness problems by restricting thread migration to only occur at barriers. We allow migration at other points in the program, as long as certain rules are followed (see Section 2.3). D-CVM also requires all static and heap-allocated thread-private data to be in shared memory. The best reported thread migration performance for D-CVM (using a reserved stack approach similar to ours) was 1.597ms on an IBM SP2 using 66.7MHz Power2 processors over a 40MB/s SP2 switch (with a stack size of 1704 bytes).

Active Threads [24] is a user level thread library that includes support for migration. One of the main goals of the Active Threads package is performance. This goal is achieved by utilizing an efficient user-level communication package based on active messages [6]. Their solution to the stack pointer problem is similar to ours. For SPARCstation-10 multiprocessor workstations connected by a Myrinet network interface, thread migration latency was reported to be about 1.1ms for 2KB stacks.

5.2. Fault Tolerance

Costa et al. [5] implement a logging and checkpoint facility to recover from single and multi-node failures on the TreadMarks [15] DSM system. They implement a two-level checkpoint mechanism. They use a lightweight logging mechanism to support single node failures and perform occasional consistent checkpoints to implement multiple node recovery. The performance results reported in [5] show that the overhead of maintaining the logs is very small. On the other hand, the re-execution needed for recovery consumed from 72% to 95% of the total execution time of three benchmark applications used (SOR, Water, and TSP). In contrast, Brazos recovery takes 0.8% of the execution time for Water even with a considerably larger dataset. Checkpoint overhead was reported as less than 2% of execution time for both Water and TSP, and around 22% for SOR. They attribute the difference to the size of checkpoints in the respective applications, which are a function of the coherence traffic exhibited by these applications.

Cabillic et al. [1] implement a consistent checkpoint facility that is similar to ours in several respects. They perform checkpoints at barrier synchronization points that are annotated by the programmer. Their checkpoint facility requires the copying of pages and page information blocks only, and does not require saving diffs. This is because their DSM system, MYOAN [2], implements sequential consistency and an invalidation-based protocol. They require that all private data be allocated in shared memory in order to expose it to the checkpoint facility, similar to our system. They implement a special checkpoint server process that requires inter-node communication to perform the checkpointing, whereas we use a checkpoint agent thread per process and communication is through hardware shared memory.

Kermarrec et al. implement a recoverable distributed shared memory system called ICARE [16]. They modified an invalidation-based coherence protocol to maintain a recovery database in volatile memory that enables recovery from single node failures. Their system replicates pages on multiple nodes to allow recovery, which in some cases resulted in improved performance since page faults were avoided.

Previous researchers have developed checkpoint facilities for Windows NT processes [10, 22], although none of these work in a distributed shared memory environment. Huang et al. implement a recovery facility for NT processes, called NT-SwiFT [10]. It includes the Winckp library that can be used for rollback-recovery of NT applications. Their system intercepts system calls and discovers areas of memory to save using standard Win32 calls. Recovery can also be performed on applications that access the network by logging network traffic. Srouji et al. [22] implemented a general-purpose checkpoint facility for multi-threaded Windows NT processes. Similar to NT-SwiFT, they redirect Win32 API calls to a set of wrapper functions that are used to save state information before calling the actual Win32 routines. This enables them to build a database of open files and other handles that need to be recreated at recovery. They describe how data segments are reserved, including static and heap-allocated memory. Checkpoint file sizes were about the same size as the process itself and checkpoint time was twenty one seconds for a 50MB process.

Finally, the Microsoft Cluster Service (MSCS) supports high availability applications such as database servers on Windows NT [7]. The MSCS can detect failures of hardware or software resources and can restart or migrate failed components. MSCS does not support rollback recovery of DSM applications, but handles situations that we do not address, including hardware fail-over.

6. Conclusions and Future Work

We have described the implementation and performance of thread migration and checkpointing mechanisms for clusters running Windows NT. The performance of thread migration was found to be competitive with other systems and an order of magnitude faster than a previously published Windows NT implementation. The checkpoint facility exhibited low runtime overhead and fast recovery times. We are currently implementing an incremental checkpointing mechanism to further reduce the overheads for large applications. We are also implementing additional techniques that combine thread migration and checkpointing for fault tolerance.

We would like to thank Karin Petersen and the anonymous reviewers for their helpful comments on earlier versions of this paper.

Brazos is available free for non-commercial use at http://www-brazos.rice.edu/brazos.

Bibliography

[1] G. Cabillic, G. Muller, and I. Puaut. The Performance of Consistent Checkpointing in Distributed Shared Memory Systems. In *Proceedings of the 14th Symposium on Reliable Distributed Systems*, September 1995.

[2] G. Cabillic, T. Priol, and I. Puaut. MYOAN: An Implementation of the KOAN Shared Virtual Memory on the Intel Paragon. IRISA, Research Report 812, March 1994.

[3] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pp. 152-164, October 1991.

[4] Compaq Corporation, Intel Corporation, and Microsoft Corporation. *Virtual Interface Architecture Specification, Version 1.0.* , 1997.

[5] M. Costa, P. Guedes, M. Sequeira, N. Neves, and M. Castro. Lightweight Logging for Lazy Release Consistent Distributed Shared Memory. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pp. 59-73, 1996.

[6] T. v. Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A Mechanism for Integrating Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, pp. 256-266, 1992.

[7] R. Gamache, R. Short, and M. Massa. Windows NT Clustering Service. *IEEE Computer*, vol. 31(10), pp. 55-62, 1998.

[8] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. L. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th International Symposium on Computer Architecture*, pp. 15-26, May 1990.

[9] R. W. Horst. *TNet: A Reliable System Area Network.* Tandem Computers, 1995.

[10] Y. Huang, P. E. Chung, C. Kintala, C.-Y. Wang, and D.-R. Liang. NT-SwiFT: Software Implemented Fault Tolerance on Windows NT. In *Proc. of the 2nd USENIX Windows NT Symposium*, pp. 47-55, August 1998.

[11] G. Hunt and D. Brubacher. Detours: Binary Interception of Win32 Functions. In *Proceedings of the 3rd USENIX Windows NT Symposium*, July 1999.

[12] L. Iftode, J. P. Singh, and K. Li. Scope Consistency: A Bridge between Release Consistency and Entry Consistency. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 277-287, June 1996.

[13] A. Itzkovitz, A. Shuster, and L. Shalev. Thread Migration and its Applications in Distributed Shared Memory Systems. *Journal of Systems and Software*, vol. 42, pp. 71-87, 1998.

[14] P. Keleher. *Lazy Release Consistency for Distributed Shared Memory.* Ph.D. Dissertation, Department of Computer Science, Rice University, Houston, TX, January 1995.

[15] P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the 1994 USENIX Winter Technical Conference*, pp. 115-131, January 1994.

[16] A.-M. Kermarrec, C. Morin, and M. Banatre. Design, Implementation and Evaluation of ICARE: An Efficient Recoverable DSM. *Software--Practice and Experience*, vol. 28(9), pp. 981-1010, 1998.

[17] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Distributed Multiprocess Programs. *IEEE Transactions on Computers*, vol. 28(9), pp. 690-691, 1979.

[18] J. Richter. *Advanced Windows.* Third Edition., Microsoft Press, 1997.

[19] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Stanford University, Technical Report CSL-TR-91-469, April 1991.

[20] E. Speight, H. Abdel-Shafi, and J. K. Bennett. An Integrated Shared-Memory/Message Passing API for Cluster-Based Multicomputing. In *Proceedings of the Second IASTED International Conference on Parallel and Distributed Computing and Networks*, pp. 146-153, December 1998.

[21] E. Speight and J. Bennett. Brazos: A Third Generation DSM System. In *Proceedings of the 1st USENIX Windows NT Symposium*, pp. 95-106, August 1997.

[22] J. Srouji, P. Schuster, M. Bach, and Y. Kuzmin. A Transparent Checkpoint Facility on NT. In *Proceedings of the 2nd USENIX Windows NT Symposium*, pp. 77-85, August 1998.

[23] K. Thitikamol and P. Keleher. Thread Migration and Communication Minimization in DSM Systems. *Proceedings of the IEEE*, vol. 87(3), pp. 487-497, 1999.

[24] B. Weissman, B. Gomes, J. Quittek, and M. Holtkamp. Efficient Fine-Grain Thread Migration with Active Threads. In *Proceedings of the 12th International Parallel Processing Symposium*, March 1998.

High-End Workstation Compute Farms Using Windows NT

Srinivas Nimmagadda, Joshua LeVasseur, Rumi Zahir
{ snimma@scdt.intel.com, jlevasse@mipos2.intel.com, rumi.zahir@intel.com }
Intel Corporation

Abstract

This paper describes our experiences in building and deploying Windows NT* based high-end workstation compute farms within the Intel engineering community. An overview of Intel's compute requirements is presented, along with the solution developed to address Intel's large compute cycle needs. The paper discusses NT's contribution to the solution, including recognized NT strengths as well as migration and deployment challenges we faced. The paper emphasizes workarounds to known issues for other user communities to leverage, but also enumerates areas in which Windows NT and the Win32 API* could be improved. Our paper concludes that despite many challenges, NT based high-end workstation farm computing is viable.

1 Engineering Computing Environment at Intel

As in many other semiconductor and computer systems companies, product development and engineering work for chip design, CAD tool development, and commercial EDA tool deployment has traditionally been carried out on Unix* based workstations. High-end workstation computing at Intel largely revolves around providing compute cycles to the various chip design projects for running a variety of computationally intensive workloads. Typical applications range from large compute-bound integer and floating-point applications (such as performance, logic and circuit simulations), to long-running jobs with very large data sets (e.g., full-chip layout verification or performance simulations of on-line transaction processing systems). Most of the jobs are compute-bound and usually run for several hours. It is not unusual for many of the compute-bound jobs to run for several days or even weeks. As a result, most of the compute cycles are spent by applications running in batch mode on a farm of hundreds of high-end multi-processor workstations. Usage models and system needs of a workstation compute farm are quite different from the more widespread file server, database client/server or web server installations typically found in large enterprises.

At Intel, hundreds of engineers submit thousands of compute and data intensive simulation jobs into the workstation compute farm every day. Jobs are typically a group of collaborating applications rather than a single executable. Within a job, applications are sequenced by scripts and communicate through files, named pipes or shared memory. Table 1 summarizes typical job parameters from an Intel workstation compute farm hosted on Windows NT.

This paper describes the challenges and experiences that we have faced in developing and deploying Windows NT based compute farms for Intel chip design computation needs.

Farm Characteristic	Typical Environment
# of Dual-processor Systems	> 100
# of Jobs per Week	> 25000
Utilization (user+system/wallclock)	> 80 %
Job Characteristics	
➢ # of Processes (per job)	1-10
➢ Memory requirements (per job)	50-500 MB
➢ Input Data Set Size (per job)	30-600 MB
➢ Job Runtime	Several minutes to several days

Table 1. Typical Workstation Farm Parameters

* Third party trademarks and brands are the property of their respective owners.

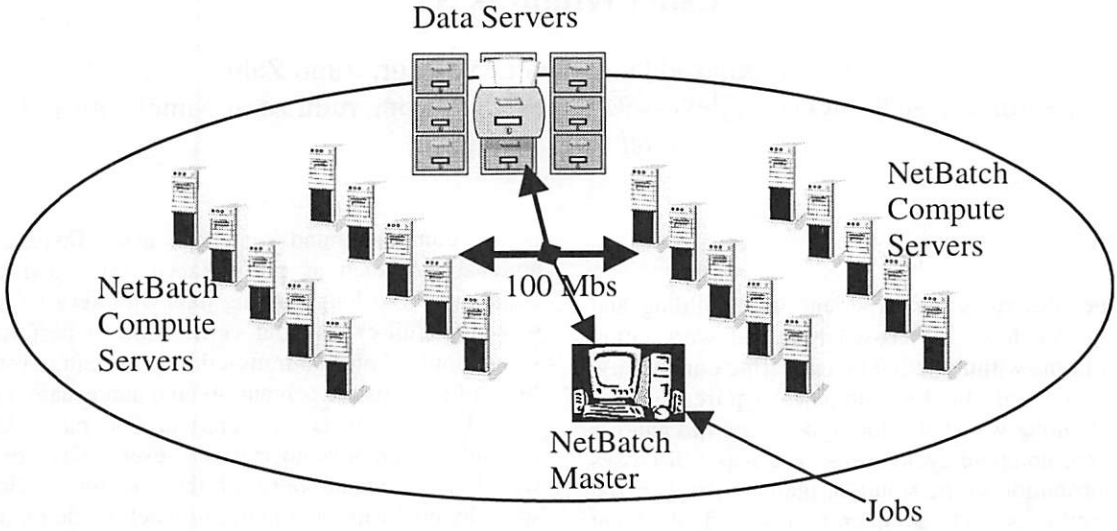


Figure 1. A NetBatch Compute Farm

2 Compute Farm Building Blocks

Our compute farms consist of hundreds of networked NT workstations built from off-the-shelf components. The machines are placed in rack and stack setting without monitor and keyboard/mouse attachments. Typical configurations include dual-processor Intel® Pentium® II 400 MHz CPUs, 512MB RAM, 9 GB local disk space, connected to data servers via 100Mb Ethernet. All machines run the Windows NT 4.0 Workstation operating system image with service pack three.

From the end user's point of view, the entire farm looks like a single virtual computer that provides on-demand compute cycles. The physical location of job execution is transparent to the user and results are reported as if the job executed locally, i.e. through output files. Typically, users develop and debug applications and scripts interactively using smaller data sets. Once debugged, the jobs run on the NT compute farm with much larger data sets, often through hundreds of iterations. Three key aspects of batch computing on Windows NT that we address in this paper are:

The Batch Computing Engine: The most important component in building a compute farm is a distributed job scheduling and execution engine. The engine provides a virtual computer view of the farm machines, transparent job execution, load balancing, and priority/quota based job-scheduling features. Section 3.1 presents NetBatch, an in-house batch engine, that we developed to address these requirements.

The Network File System: To provide batch jobs with the illusion of a single virtual computer, a unified run-time environment and name space are required at three different levels:

- a high-performance unified file system (data) view,
- a unified secure data access mechanism, and
- a unified user profile (path, environment variable and application settings).

Section 3.2 presents various file system and data access mechanisms used by our compute farm. These technologies include Samba* (public domain), DFS* (from Transarc) with DCE* (from Gradient), and NTFS*/SMB* servers (from Microsoft).

Application Development and Porting Issues: Most compute jobs consist of multiple collaborating applications that are glued together with perl or tcsh scripts. Porting of these legacy scripts to NT has often required more effort than actually porting the applications themselves. In cases where scripts are heavily dependent on idiosyncracies of Unix commands (such as ls, cat, grep, awk, sed, etc.), porting of legacy scripts just requires too much effort. Another paper in these proceedings [MTEX], describes a solution that allows us to run applications on NT while the bulk of the scripts execute on Unix. In Section 3.3 (in this paper), we discuss our experience using the public domain Cygnus utilities [Noer 98] on NT directly, and we identify key technical short-comings of NT's native scripting support.

3 Challenges and Solutions

3.1 NetBatch: A Distributed Batch Computing Engine on Windows NT

Figure 1 shows a high level view of a NetBatch compute farm. A NetBatch pool is a network of NT workstations, each running a NetBatch server process, which communicates with a pool master. The NetBatch Server process monitors the individual machine load and resources, and provides the seamless job execution environment. Each NetBatch pool has one pool master controller responsible for providing the user interface (for job acceptance, control, etc.), load-balancing and job scheduling capabilities. The load balancing service on the master collects load and machine availability information from the machines within the cluster, and selects least loaded machines for job execution. The scheduler service applies a set of criteria (based on priorities and user quotas) to the queued batch jobs to select the best job for execution.

The following sections highlight some of the key areas of our batch engine implementation:

- **Job Tracking and Management:** how NetBatch executes and tracks batch jobs with multiple child processes.
- **User Impersonation:** how NetBatch provides transparent execution of single or multi-process jobs under the exact user credentials and environment that existed at the time of submission.
- **Job Scheduling and Suspension**: how NetBatch load balances the compute farm by dispatching jobs to idle or least loaded machines, and permits suspension of running jobs.
- **Batch Computing on Interactive Desktop Workstations**: how NetBatch uses idle desktop workstation compute resources.

3.1.1 Job Tracking and Management

To carry out its functions, NetBatch must track batch jobs that spawn multiple child processes. For example, to track the resource usage of a batch job, and to suspend or terminate the job in response to a user request requires knowledge of a complete process tree of the job. The first approach we considered used a technique (similar to one implemented in the Microsoft SDK `tlist` command) to identify parent-child relationships by walking the kernel process data structures to construct the process tree. However, termination of an intermediate process splits the process tree and complicates group tracking. Another technique that we explored attaches a dummy object to the parent with a child inheritable handle. Then all processes containing a handle to the dummy object can be counted as part of the same process group. This method is also infeasible because applications can disable handle inheritance across process creation. We tried using session identifiers to provide another possible solution because each batch job impersonates the user (submitter of the job). This technique, however, does not work in scenarios where a job requires true process group support from the OS. Examples of this include application initiated process group control where a process can switch between groups and other accounting purposes. This prompted us to pursue a native NT solution with Microsoft, which has resulted in a new Windows 2000 feature known as job object [Solomon 98]. A job object is a new kernel object that provides capabilities such as process grouping, resource usage collection (on a group basis), and enforcement of process resource limits. Our conclusion is that job object feature is well suited for job tracking and management under Windows 2000, while any of the above workarounds provide reasonable solutions with restricted scope under NT 4.0 release.

3.1.2 User Impersonation

All processes in a batch job need to run with the exact user credentials of the person submitting the batch job. NT impersonates a user with the `LogonUser()` and `ImpersonateLoggedOnUser()` APIs. However, to use the `LogonUser()` API, the NetBatch Server requires the user's clear text password. There could be a significant time lapse between the NetBatch job submission and machine availability in the pool for execution of the job, during which the password may change.

To provide a clear text password to LogonUser(), NetBatch could collect the passwords at the time of job submission and securely store them until job execution. Alternately, all passwords could be collected in advance and stored in a two-way encrypted user password database for later retrieval to use with the `LogonUser()` API. However, maintaining a two-way encrypted database poses security concerns and also creates administration and password synchronization issues. This prompted us to investigate further and resulted in the creation of a password-less impersonation mechanism in NT. We enhanced the winlogon local security authority system with our representative DLL, which permits the NetBatch batch job to login without a password. However, this approach has a limitation that the impersonated process cannot access SMB network files.

In a Unix environment, user impersonation is a simple task with the `setuid()` system call, where the administrator (root) can impersonate a user for executing the batch job. Due to these reasons we feel that a true, trusted delegation-based impersonation capability would be extremely useful in NT.

3.1.3 Job Scheduling and Suspension

Load balancing between machines is an important factor in improving the overall throughput in the NT batch farm. Choosing the machine with the least-loaded CPU and the most suitable free memory, page file, and free temporary disk space configuration is an important function to optimize in a farm environment. In Windows NT, these indicators can be obtained from the system supplied performance counters. The NetBatch server periodically scans these counters and reports these machine availability indicators (CPU load, free memory, disk space, etc.,) to the NetBatch master for making best scheduling decisions. Since NT doesn't have a native performance counter that gives CPU load, we had used a formula similar to the one used to compute CPU load factor in Unix operating systems. This load factor is computed as the weighted moving average of the number of ready to run threads over 15 minutes. The formula for the load factor is $(10*N_1 + 4*N_5 + N_{10})/15$; where N_x is the average number of ready to run threads in the system during the last x minutes. It will be an interesting research topic to create a general-purpose machine availability indicator(s) based on available free CPU, memory, page file, and free disk space indicators, and further quantifying job run-time requirements using these indicator(s).

One of the features of our batch engine is to suspend execution of batch jobs on demand. Suspension of the job requires suspension of each process in the batch job. However, Windows NT doesn't provide a process suspension feature, although it can suspend individual threads in a process with the `SuspendThread()` `API`. However, this Win32 API function requires a handle to each thread in the process. Using the performance counters, it is easy to enumerate the thread IDs in each process. Win32 doesn't provide a mechanism to convert a thread ID to a thread handle! We can obtain a process handle given a process ID, but there is no way to suspend a process. This deficiency in NT is worked around using process debug APIs to obtain the thread handles. All threads are suspended resulting in process suspension. It would be very useful if the Win32 API supported a built-in process suspension mechanism, or exposed the internal APIs to convert a thread ID to a thread handle.

3.1.4 Batch Computing on Interactive Desktop Workstations

Intel deploys large numbers of powerful NT desktops for interactive desktop use. One of our goals is to utilize the unused computing bandwidth of these desktops during off-hours. The following are some key challenges we have encountered and addressed in providing farm computing using interactive desktops.

Detecting when a machine is being used interactively is an important component for integrating a user's desktop machine with the NT compute farm. NetBatch uses the Windows NT `SetWindowsHookEx()` API to setup hooks that monitor the keyboard and mouse activity and thus detect the presence of interactive users. After detecting a user, NetBatch takes appropriate intrusion avoidance actions including: preventing new batch jobs from beginning while the user is present, suspending already running jobs, and optionally terminating and resubmitting jobs to a different machine. While detecting the presence of console users is easy, detecting remotely logged-in users, such as logins from home using *telnet*, *rcmd*, *rsh* services, is a challenging task. We have identified a workaround for this by polling the process tree and identifying the unique users every few seconds. However, we found this type of polling is expensive when combined with the batch server's other tasks. We feel this is an important counter that the NT core OS should add to the list of performance counters. Without OS support, this is an expensive and sometimes inaccurate task to perform at user level.

Several of our batch jobs run for days to weeks. One of the capabilities we would like to have in the core OS is support for application checkpoint and migration from one machine to another. This feature would allow the batch engine to free up a desktop machine when the user resumes interactive work. At present, we have solved this problem using a combination of methods. The first approach temporarily suspends the batch job when the interactive desktop user is present and restarts the job when user leaves. The second approach suspends the job for a period of time, and if the interactive user is still present, terminates the job and restarts it on a different machine.

This impacts the overall throughput and turn-around time for the jobs. Another approach requires application-level state checkpointing and resumption. A group at Intel [Srouji 98] developed a transparent checkpointing method on top of NT, but it places limitations on the type of applications that can be checkpointed. Middleware approaches require

application source modification, or recompilation, or re-linking. This is not favourable in a typical EDA environment with integrated suites of applications from external and internal sources. Despite this, most middleware approaches also have limitations on check-pointing multi-process applications, or applications with IPC and other process dependencies. Our conclusion is that complete process checkpoint and migration is difficult without support from the core OS.

Disabling application popup error dialog boxes on a per process basis is very important. This is very useful especially if the batch jobs run on interactive desktops. A critical error in a batch program should not display a dialog box on the interactive desktop. Apart from interactive users not liking the popup display, it suspends any running batch jobs until someone hits OK or Cancel on the dialog box. Using the existing NT API `SetErrorMode()`, it is possible to disable pop-ups for batch applications.

3.1.5 Batch Engine Performance

Our NetBatch masters run on either Unix or NT platforms and manage NT farm servers. Our master code is mostly algorithmic (rather than system) code and the performance numbers for masters were roughly equivalent on both Unix and NT platforms. Master takes about 21% CPU for controlling and load balancing a cluster of 1000 machines. Average NetBatch overhead for managing a job through its life cycle (queuing, scheduling, providing execution environment and utilization metrics logging) is 400 milliseconds. The overhead on NT server is about 110 ms for starting the job, while 40ms on a comparable Unix machine. Of the additional 70ms overhead on NT, about 30ms were spent simulating fork() like API, 20ms were spent simulating the password-less (`setuid`) style code, and the rest for user environment translation and other NT specific overheads. This additional overhead is negligible, as it is a one time task required for launching a job that could run for long time. While the job is running, job monitoring (such as suspension, altering priorities, termination, etc.,) on NT has additional overheads due to lack of process groups and resource usage information. This was observed as additional 0.5-1% CPU consumed by our NT server during the job runtime for executing workarounds described in section 3.1.1 and section 3.1.3.

3.2 The Network File System

To make a user job run on any machine in the compute farm, we needed a uniform environment at three levels: a uniform file system space, a uniform data access control mechanism, and uniform user login profiles. Our batch compute jobs cover a wide variety of file access patterns: read and write to small data files, sequential read or write to multi-gigabyte data files, random access to large databases, repeated demand loads of executable pages, and reuse of the files. As a result, any data access mechanisms have to provide excellent network file system performance, and scale to facilitate concurrent data accesses to hundreds of batch farm clients.

Intel's support structure favors centralized file servers for easy administration and maintenance operations such as backup, retrieval, and project resource partitioning. Locating file data in central servers, and making the user login access profiles uniform across all the machines (interactive and batch) is the first step in providing a unified job run-time environment.

Along with NT farms, we have a large install base of Unix workstations that use NFS and AFS file systems. Applications in both NT and Unix environments need access to common data input files. To serve these cross-domain requirements we have considered three types of file or data sharing approaches: Samba, DFS, and native SMB (NT file servers).

Samba: Samba is an open source product that acts as a gateway between Unix file systems and SMB protocol based file clients. The Samba server runs on a Unix machine and can integrate file systems from different servers and export them to NT clients as a single share. It has the advantage of providing access to the unified name space of the Unix file system, but suffers from scalability and performance issues. We partially solved the performance problem by replicating the gateways and thus reducing the clients per gateway ratio. In our environment, logic simulation jobs required a ratio of one Samba gateway (running on a powerful Unix workstation) per 100 clients, where each client accessed 10-15MB of data over a period of a couple hours. The Samba solution did not satisfy the needs of architectural performance simulation jobs, which often have a large file working set (gigabytes of trace files), and demand load many pages from the executables.

DFS: DFS is a distributed file system from Transarc, which offers a global file name space shared by all clients, hides the disjoint nature of the DFS servers, offers local disk caching for improved performance, and provides ACL based file access controls. DFS file systems offer great flexibility and performance. In our environment we have chosen DFS for our data intensive compute farms. Initial versions of DFS lacked support for multi-user access on NT clients and experienced occasional stability issues. We worked with DFS/DCE

vendors (Transarc and Gradient) to add the multi-user features to DCE2.2 and DFS. This latest release also addressed several stability issues that existed in earlier releases of DCE and DFS.

SMB: We also tested the use of NT file servers with SMB based NT clients. The NT file servers provide access to their files through shares, which the NT client can access through UNC paths, or by mapping the share to a drive letter. UNC is not popular as it lacks support for standard file system operations such as changing or setting the current working directory. Microsoft Dfs offers an integrated name space solution for SMB clients, but it was dropped due to reliability problems with the earlier version. The drive letter approach was the only choice left for data access on NT clients but it also has a few issues. The drive letters are established globally throughout the OS, rather than on per user or session basis, which restricts the number of concurrent mappings, and opens security holes in a multi-user environment. NT also leaves stale mounts when a user login session terminates without unmapping a drive.

SMB suffers from reliability and scalability problems. Its reliability suffers from packet time-outs, which cause application level errors. Application binaries stored on a remote NT file server, when executed on a NetBatch compute farm machine, often experience abnormal program terminations due to packet time-outs when the OS attempts to demand load an unmapped code page. This problem can be solved by making a local copy of the image file, either by explicitly copying the file, or by setting a bit (/swaprun:net) in the image file which triggers the NT loader to make a local copy in the pagefile prior to execution. The main disadvantage of this approach is reduction in throughput and bottlenecks at the file server, especially when binary files are large and many of them execute concurrently on hundreds of farm machines. This problem has been addressed in other environments, such as on NFS based Unix clients, where the process execution is kept on hold state until normal communication with the file server is restablished. This mechanism isolates long running applications from transient network data access failures.

Filesystem Reliability Comparison: We attempted to compare the reliability of DFS, SMB with an NT server, and SMB with a Samba server exposing NFS file systems, within our compute farm environment. The workload was based on architectural performance analysis of gigabytes of compressed trace information from an online transaction processing benchmark. The workload is I/O bound, although a fair amount of CPU time is required for decompression. The data fetch and the data decompression are pipelineable, if the file system supports a streaming prefetch mechanism. The reliability experiments executed in the standard compute farms during weekends. The compute farms continuously handle a load under indeterminate network conditions. But restricting the data collection to weekend runs helped isolate the file servers from the spurious network traffic due to interactive users. The performance data was captured while executing one, two, five, and ten simultaneous jobs in a NetBatch cluster. All jobs requested the same file information from the server. Executables were located within the DFS filesystem, to protect the jobs from packet timeouts during demand loads of code pages. Table 2 tabulates the results. DFS successfully handled the load. SMB with the NT fileserver suffered from data packet timeouts under more intense loads. The Samba server and NFS servers were too loaded to return requests within the packet grace period; a single Samba server is the point of entry for all architecture related NT clients into the NFS file system.

SMB Summary: We suspect that poor caching by SMB causes an increased load on the network, which raises the probability of a packet time-out and thus a loss in reliability. SMB's reliability problems limit its scalability in our batch computing environment.

We abandoned the SMB/NT file server solution due to the above reasons, and due to the lack of robust cross-domain (NT to Unix client) capabilities. There are two key areas which NT must address to make the native SMB file system an acceptable solution in a typical engineering computing environment: i) improve reliability and scalability, and ii) improve client side

Filesystem	1 job	2 jobs	5 jobs	10 jobs
DFS	8.71 hours/job	8.0 hours/job	9.95 hours/job	10.85 hours/job
SMB with NT fileserver	8.45 hours/job	9.68 hours/job	failure	failure
SMB with Samba and NFS	failure	failure	failure	failure

Table 2. Filesystem reliability for a disk intensive workload (several gigabytes).

share mapping mechanisms (improved UNC and a superior approach to the multiple drive letters).

3.3 Application Development and Porting Issues

Many of our engineering applications were originally developed and deployed on Unix systems. While many solutions exist that emulate Unix services on NT such as OpenNT* [Walli 97], NutCracker*, and Cygwin* [Noer 98], we decided to convert our applications to NT's rich native Win32 API in most cases. For the large part this has been straightforward, especially since many of our applications are non-graphical in nature. In some instances, however, we have been unable to convert applications to the Win32 API, and in these cases, we use the Cygwin libraries and Cygnus utilities. We use these public domain tools for scripting, and for porting code that requires the use of the UNIX fork() system call, which is not provided by Win32 API. It seems to us that these two capabilities would be good candidates for future Windows NT or Win32 API extensions.

3.3.1 Scripting Solutions

As stated earlier, many of our batch jobs string applications together using scripts. As a result, our design automation team chose to use Cygnus utilities and public domain perl/tcsh packages to support our batch mode scripting needs. While we have encountered several stability issues with the cygwin.dll b.17 version (related to concurrency and multi-user usage of the GNU commands), migration to cygwin.dll b.20 seems to have addressed the majority of our issues. Currently it looks like we will continue to use these public domain scripting engines, since the native Windows NT scripting command shell (cmd.exe) only provides very limited capabilities. For instance, it is missing regular expression parsing and string processing capabilities, and it also lacks an interface with the Win32 API. To provide a batch friendly environment, Windows NT needs a more powerful, customizable and rich native shell environment. Another fundamental shortcoming for scripting on NT is that the Windows NT executable loader only recognizes files with certain extensions as executables. This problem can be solved if the NT loader is modified to recognize scripts using the file execute attribute, and then use the first line of the script to determine which scripting engine to invoke.

3.3.2 Fork System Call

In the traditional Unix environment, the fork() system call has been conveniently used to replicate the process state and address space. Our simulators protect the state of long-running simulation jobs by periodically creating a child image. The parent image waits for completion of the child image. If the child image encounters live-locks or run-time errors, the parent simulator will terminate the child image and fast-forward the simulation beyond the condition causing the problem. The isolated address space of the child simulator insulates the parent from any of the child's errors and memory leaks. The child also has access to another 2 gigabyte address space for data collection. While the Windows NT kernel does provide process address space and some state cloning capabilities, the Win32 API stops short of providing a complete process clone interface, even for non-graphical "console" applications. The NT thread paradigm does not replace the benefits of the fork paradigm, for a faulty thread can damage the entire simulator. The one-for-one cloning of a process address space, along with associated operating system state semantics, is a powerful and much needed Win32 API.

4 Recognized NT Strengths

With the exception of earlier workarounds, NT provides many other attractive features, which we used to develop a robust and simplified NetBatch engine, some of which are listed here. The NetBatch components are highly multithreaded, using native kernel threads to handle intrusion detection, job control, resource monitoring, server communication, etc. Various components utilize asynchronous procedure calls (APC) and mail slots to create a robust communication infrastructure. Performance counters provide a central place to get a wide range of machine resource indicators and process metrics used for optimal job scheduling decisions. Process priority classes provide a way to utilize machine cycles with minimal intrusion with other tasks and improve latency for (with real time priority) critical tasks. SCM (Service Control Manager) provides a good way for installing and managing long running NetBatch services on hundreds of farm machines. The NetBatch engine utilizes the event log for debug and error analysis. Apart from these core OS facilities, Visual Studio (MSVC) Integrated Development Environment (IDE) and Win32 SDK provides excellent software development, debugging, and maintenance environment.

* Third party trademarks and brands are the property of their respective owners.

5 Related Work

Condor [Bricker 91, Litzkow 88] is a high-throughput batch-computing environment for networked workstations. It is sensitive to the needs of desktop users and makes use of unused desktop cycles. It uses flexible match making algorithms [Raman 98] to find the best machine to execute each job. Condor also provides transparent checkpointing and migration of batch jobs on intrusion detection. Some limitations of Condor include the unavailability of a complete batch solution on NT, and limitations on check pointing most multi-process, communication, and I/O intensive applications in the typical high-end computing environment. The distributed nature of Condors *schedd* makes enforcing user or group scheduling policies and fair share allocation difficult. The gateway architecture for multi-clustering makes implementation and managing cross-site policies easier, but results in poor scalability and reliability due to the complexity of the remote scheduling protocols.

LSF (Load Sharing Facility) from Platform Computing Inc., is based on the Utopia architecture [Zhou 92]. The LSF suite has a complete set of job scheduling and load-balancing features. An NT port of LSF is also available. Some limitations of LSF on NT are the lack of transparent authentication and data access support, and lack of good cross domain (Unix-NT) seamless computing support.

Microsoft's Wolfpack [MSCS 98] provides clustering extensions to the Windows NT operating system that promise scalability and availability of servers in a mission critical enterprise environment. It has limited load-balancing features, but does not provide job-scheduling capabilities. Wolfpack can be used to scale and load-balance data servers, while the distributed batch engine described in this paper is able to schedule and load-balance jobs across NT workstation compute farms.

6 Summary

Intel's engineering community generates many compute intensive jobs, suitable for batch execution, which may sustain execution times of hours to weeks. Users see the batch pool as a large virtual computer that provides on-demand compute cycles while seamlessly executing the user jobs. Three key components necessary to establish the homogenous virtual computer is a batch compute engine, a unified job run-time environment, and application infrastructure to manage large workloads. The batch compute engine manages workstation resources, distributes load across many compute nodes, controls the jobs, and manages the job run-time environment. The unified job run-time environment ensures consistency between the interactive and batch compute environments, provides a global file name space, enforces a global security mechanism, and propagates application configurations.

As this paper describes, NT offers solutions (either direct capabilities or workarounds) for most of the large compute farm issues. At Intel, we successfully use NT compute farms for some of our mission critical needs. Our experience has shown that the overall NT environment stability is comparable to that of the Unix environment. Our conclusion is that NT is a viable choice as an OS to deploy for the infrastructure of a large compute farm, and offers tremendous cost/performance advantages to RISC/Unix workstation solutions. In this paper we have presented several challenges and solutions that other NT user communities can leverage, and presented areas in which Windows NT and Win32 API could be improved further.

The following lists offer some suggestions for improving Windows NT and the Win32 API.

Operating System Related Improvements:

- Provisions for secure user impersonation without using clear text passwords.
- Addition of a fork() system call.
- Improved shell support for scripting.
- Operating system support for process suspension, check pointing and restart.
- Support for a true multi-user execution environment in Windows NT.

File System Related Improvements:

- Improve network file system reliability and scalability.
- Support for a unified name space and better client side share access features.

Future improvements to our batch engine described in this paper include better intrusion detection for local and remotely logged-in users, improved scheduling schemes, and better support for multi-cluster capabilities.

Acknowledgements

Thanks to Tae Paik, Eldon Chan and other members of the corporate NT engineering computing program for their great encouragement and constant support for this effort. Thanks to Raghu Krishnamurthy, Ty Tang, and Mike Hester for their contribution in developing the

batch engine on NT during early 97 and Dave Liebson for his contributions in addressing several farm issues. Thanks to Drew Hess and Tom Willis for providing valuable technical feedback on this paper. The progress we have made couldn't have been possible with out the help from many other members of our IT NT team and great co-development effort from our ever demanding internal customers.

References

[Bricker 91] Allan Bricker, Micahel Litzkow, and Miron Livny: "Condor Technical Summary", University of Wisconsin – Madison, 1991.

[Litzkow 88] M.J. Litzkow, M. Livny, and M.W. Mutka. "Condor – A Hunter of Idle Workstations". In Proc 8th International Conference on Distributed Computing Systems, San Jose, California, June 1988.

[MTEX] T. Tang, V. Lal, and S. Krishnapura, "MTEX: A Bridge For Migrating CAD Design Environment From UNIX To NT", Usenix Windows NT Symposium, 1999.

[MSCS 98] White paper on "Windows NT Load Balancing Service (Wolfpack) technical overview", *http://www.microsoft.com/ntserver/ntserverenterprise/techdetails/prodarch/wlbs.asp* and *clustarchit.asp.*

[Noer 98] Geoffrey Noer, "Cygwin32: A Free Win32 Porting Layer for UNIX Applications", Proceedings of 2nd USENIX Windows NT Symposium, Seattle, Washington, August 3-4, 1998.

[Raman 98] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed Resource Management for High Throughput Computing, *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, July 28-31, 1998, Chicago, IL.

[Solomon 98] David Solomon, "Inside Windows NT", Second Edition, Microsoft Press.

[Srouji 98] Johny Srouji, Paul Schuster, Maury Bach, and Yulik Kuzmin: "A Transparent Checkpoint Facility on NT", Proceedings of 2nd USENIX Windows NT Symposium, Seattle, Washington, August 3-4, 1998.

[Walli 97] Stephen Walli: "OPENNT™: UNIX Application Portability to Windows NT™ via an Alternative Environment Subsystem", Proceedings of USENIX Windows NT Workshop 1997, Seattle, Washington, August 11-13, 1997.

[Zhou 92] S. Zhou. LSF: Load sharing in large-scale heterogeneous distributed systems. In Proc. of Workshop on Cluster Computing, 1992.

[Zhou 92] S. Zhou, J. Wang, X. Zheng, and P. Delisle. Utopia: A Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems, Technical Report CSRI-257, University of Toronto, Toronto, Canada.

High-Performance Distributed Objects over System Area Networks

Alessandro Forin, *Microsoft Research*
Galen Hunt, *Microsoft Research*
Li Li, *Cornell University*
Yi-Min Wang, *Microsoft Research*

Abstract

In this paper, we describe an approach to build high-performance, commercial distributed object systems over system area networks (SANs) with user-level networking. The specific platforms we use in this study are the Virtual Interface Architecture (VIA) and Microsoft's Distributed Component Object Model (DCOM). We give a detailed functional and performance analysis of DCOM and apply optimizations at several layers to take full advantage of modern high-speed networks. Our optimizations preserve the full set of DCOM features including security, alternative threading models, and Microsoft Transaction Server (MTS). Through extensive runtime, transport and marshaling optimization, our system achieves round-trip latencies of 72 microseconds for DCOM calls and 174 microseconds for MTS calls, and an application bandwidth of 86.1 megabytes per second. We also examine the performance gains in real applications.

1. Introduction

With the explosive growth of the Internet and advances in high-speed networking, distributed computing is becoming a predominant programming paradigm for building mission-critical applications. In recent years, research and development efforts along three lines have significantly changed distributed computing. First, researchers have long observed that the software overhead in the communication protocol stacks accounts for a significant fraction of the end-to-end transmission delay. The intervention of the operating system in the critical path and repeated copying of intermediate buffers hinder the delivery of order-of-magnitude improvements in raw network speed to applications. Several research projects have proposed and implemented fast networking protocols and interfaces that allow user-level access to high-speed networking devices. Examples include Cornell U-Net [V95], Illinois Fast Messages (FM) [P97], Princeton Virtual Memory-Mapped Communication (VMMC) [B94], etc.

Another trend in distributed computing is the increasing popularity of server clusters. By connecting a number of relatively inexpensive machines with high-speed System Area Networks (SANs), such configurations offer a cost-effective approach to achieving high performance and availability. The Virtual Interface Architecture (VIA) [V97], proposed as an industrial standard user-level networking architecture, promises to deliver much of the raw power of SAN directly to applications.

The third trend is object orientation. Distributed object systems, such as Distributed Component Object Model (DCOM) [B98], Common Object Request Broker Architecture (CORBA) [C95], and Java Remote Method Invocation (RMI) [W95], extend the benefits of object-oriented programming to the networked environment. These systems provide an infrastructure that hides the details of low-level communication mechanisms and presents a higher-level abstraction to programmers to simplify distributed programming.

Just as high-speed networks shift the performance bottleneck to protocol stacks, commercial user-level networking shifts the bottleneck to distributed-object infrastructures. In this paper, we use DCOM over VIA as an example to investigate the design issues in providing high-performance distributed object systems over user-level networking. The target application environments are physically secure server clusters consisting of homogeneous machines connected by a high-speed system area network. Our goal is to implement a set of software modules that can be integrated into the existing DCOM infrastructure. These modules will be loaded for inter-server communications within a cluster, while client machines outside the cluster continue to contact the server machines through traditional protocol stacks running over traditional networks.

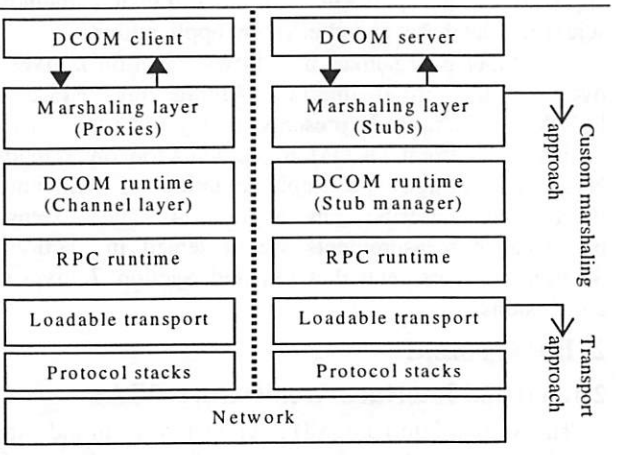


Figure 1. Layered architecture of DCOM.

Figure 1 illustrates the high-level architecture of DCOM. The marshaling layer packs method-call parameters and converts data formats between machines with different architectures. The DCOM runtime maintains object and interface identities, enforces DCOM-level access control, and supports different threading

models. The RPC runtime manages thread pools, message multiplexing, interface bindings, and authentication. At the loadable-transport layer, each transport exports a common interface to encapsulate network protocol-specific details from the runtime layer. Loadable transports reside in dynamic link libraries (DLLs) separate from the runtime DLLs.

Figure 1 suggests several approaches to run DCOM over VIA with different tradeoffs. The *custom marshaling approach* [C98][Ma98] uses a custom marshaling layer to run DCOM applications directly on VIA, bypassing all runtime support from DCOM and RPC. This approach can deliver almost all of the raw VIA performance, but does not support the full set of DCOM features. The *transport approach* adds a new loadable transport module that encapsulates all VIA-specific details. The new transport transparently supports all DCOM and RPC applications, however the RPC runtime seriously limits potential performance gains [Z98].

In contrast, our system, *Millennium Falcon*, leverages the existing DCOM runtime layer to support the full set of DCOM features. Optimizations for VIA are implemented in the loadable transport, RPC runtime, and marshaling layers. At the loadable transport layer, we exploit RPC semantics to perform efficient flow control. At the marshaling layer, we achieve zero buffer copies by exposing scatter-gather I/O to proxies and stubs. At the RPC runtime layer, we implement a new binding-handle module to improve DCOM critical-path performance. We address both latency and bandwidth issues. The former is important for the common case of method calls with small-size parameters. The latter is important for applications that perform bulk data transfer, including scientific, database, and checkpoint applications.

The paper is organized as follows. Section 2 gives an overview of Virtual Interface Architecture, RPC, and DCOM. Section 3 presents a layered performance analysis of current DCOM implementation on Windows NT. The design and implementation of Millennium Falcon are described in Section 4, and extensive performance measurements are presented in Section 5. Section 6 covers related work, and Section 7 gives the conclusions.

2. Background

2.1. Virtual Interface Architecture (VIA)

The Virtual Interface (VI) Architecture is an industrial, user-level networking architecture for high-bandwidth, low-latency, and low-overhead communication [V97]. In contrast to traditional network architectures where the operating system virtualizes network hardware into a set of logical endpoints, the network adapters in the VI architecture take over the task of endpoint virtualization, data multiplexing and transfer scheduling to reduce OS involvement. Each VI represents an endpoint with a direct, protected interface to network hardware. A process may acquire multiple VIs exported by one or more network adapters.

A *VI consumer* is typically an application program that uses VIs through some communication facility such as sockets. The communication facility usually loads a *VI user agent* library supplied by the hardware vendor to abstract the details of the underlying VI provider. The *VI provider* consists of a network interface controller that implements the VIs and directly performs data transfer functions, and a *VI kernel agent* that performs setup and resource management functions. The VI consumer registers the memory buffer with the VI provider before submitting a data transfer request. This allows the consumer to reuse the registered memory buffer for subsequent data transfers to avoid the overhead of duplicate page lock/unlock operations and address translations.

Each VI consists of two *work queues*: a *send queue* and a *receive queue*. A request from the VI consumer to send (or receive) data is posted on the send (or receive) queue as a *descriptor*, which contains all the information needed by the VI provider to process the request. A *doorbell* associated with each work queue notifies the network adapter that a new descriptor has been posted. The VI provider asynchronously processes the descriptors and marks them with status values upon completion.

2.2. Remote Procedure Call (RPC) and Distributed Component Object Model (DCOM)

Microsoft RPC is an implementation of the DCE RPC [D95] specification. It uses the Network Data Representation (NDR) format for data marshaling in heterogeneous environments. Typical RPC client and server applications are structured as follows. The server application specifies a protocol in an RPC API call, which loads the corresponding transport DLL and creates a communication endpoint. The server invokes another API to register the RPC interfaces on which it expects to receive method calls. An *interface* consists of a set of functionally related method calls. Each RPC interface is identified by a 128-bit Universally Unique Identifier (UUID) called an *Interface ID* (or *IID*), and is specified in an Interface Definition Language (IDL) file. Once the server is ready to receive calls, it constructs an RPC *string binding*, which contains sufficient information to identify the server endpoint. The string binding is propagated to the client through a naming service or some other means. The client constructs a binding handle from the string and makes RPC calls through the handle.

DCOM extends DCE RPC to support the notion of objects with multiple interfaces and to provide a mechanism for server activation. At the marshaling layer, the current DCOM implementation reuses MSRPC NDR code for data marshaling and augments it with the support for marshaling *object interface pointers* into object references. An *object reference* contains sufficient

information to locate a unique object interface instance within a unique server endpoint. On the client side, DCOM inserts a *channel layer* between the marshaling layer and RPC runtime, as illustrated in Figure 1. Each channel object encapsulates an RPC binding handle and manipulates the DCOM-specific part of each RPC packet. The counterpart of the channel object on the server side is the *stub manager*, whose main task is to dispatch each incoming call to its target stub.

Typical DCOM client-server interactions proceed as follows. The client application invokes the *CoCreate-InstanceEx()* API to either activate a server application or connect to a running server process. The client specifies a *Class ID* (or *CLSID*), which is the UUID of an object class, and the IID of the interface to which it is requesting a pointer. As a result of this activation, the server process creates an object instance of CLSID and (logically) returns to the client a pointer to the object's IID interface. The client can then invoke methods through that pointer as if the object resides in the client's own address space. When the client needs a pointer to another interface of the same object instance, it makes a *QueryInterface()* call on the current interface pointer and supplies a new IID. In Section 4, we will describe in more detail how current DCOM implementation provides this object-oriented abstraction on top of MSRPC, and discuss its impact on performance.

3. Where Do The Cycles Go

We present here an analysis of the overhead distribution among layers in the current DCOM implementation over TCP. Measurements were obtained by analyzing DCOM/RPC source code and intercepting layer-crossing calls through binary instrumentation. The analysis identifies performance bottlenecks and predicts the effectiveness of candidate optimizations. Our measurement setup consists of a pair of Gateway 2000 E-5000 PCs, each with a 333 MHz Pentium II CPU and 256 Mbytes of RAM, and running Windows NT Workstation 4.0 with Service Pack 3 (NT4 SP3). The machines are directly connected through a pair of 1.25-Gb/s GigaNet GNN 1000 adapters. The test program, *PingPong*, invokes a DCOM call to send buffers of the same size back and forth. The buffer size is specified as the first method parameter followed by the pointer to the buffer specified as an `[in,out]` parameter. The standard, compiled proxy/stub code generated by the Microsoft IDL (MIDL) compiler is used.

Figure 2 illustrates the round-trip overhead distribution across different DCOM data sizes. All numbers are averaged over 1,000 runs. In both plots, at the top layer, the marshaling overhead results from gathering call parameter data into the RPC buffer. In the middle, the runtime overhead represents the data-size-independent portion of DCOM and RPC processing. At the bottom, the transport overhead includes the execution times of the RPC loadable transport, the TCP protocol stack, and the actual wire time.

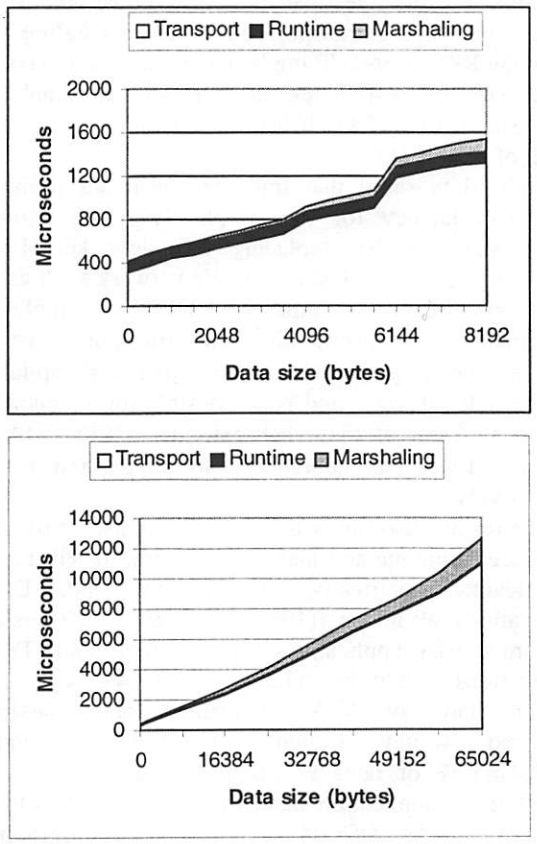


Figure 2. Round-trip overhead distribution of DCOM over TCP: small data size (top); large data size (bottom).

Round-trip Latency (null call)	413 µs
Round-trip Latency (64KB)	12.6 ms
Max. DCOM Bandwidth	11.4 MB/s
DCOM/RPC Runtime	100 µs
Marshaling Time	7 µs + 18 µs/KB + 34 µs/KB above 50K
Transport Time	313 µs + 110 µs/KB + 260 µs/5840 bytes

Table 1. Summary of DCOM-over-TCP performance numbers on GigaNet GNN 1000.

Table 1 summarizes the essential numbers from the two plots. The average round-trip latency for DCOM calls without any parameters is 413 µs. At the other extreme, the latency for close to 64KBytes is 12.6 ms. (The Maximum Transfer Unit, MTU, for the GigaNet cards is 17 bytes below 64KB.) The maximum bandwidth of 11.4 megabytes per second is achieved at a data size around

20KB. The fixed DCOM and RPC runtime overhead is approximately 100 µs.

Table 1 also gives the analytical equations for the curves that fit the transport time and marshaling time. Since the RPC-over-TCP implementation uses a maximum buffer size of 5840 bytes, there is an additional fixed overhead for every 5840 bytes (the discontinuity in the top graph of Figure 2).

Both plots show that transport overhead dominates round-trip latency for the simple *PingPong* program, which suggests that replacing the slow kernel-mode protocol stack with a fast user-mode network such as VIA can significantly improve DCOM application performance. Specifically, the transport overhead accounts for 313 µs (76%) of the 413 µs round-trip latency for the null-call case, and is responsible for 11 ms (87%) of the 12.6 ms at the other extreme. With VIA, the transport times can be reduced to 30 µs and 1.4 ms, respectively.

The results also imply that once the transport overhead is reduced, runtime and marshaling overhead will become significant. Specifically, at the low end, DCOM applications would be ((100+30)-30)/30 = 333% slower than raw VIA applications. At the high end, DCOM applications would be ((12.6-11+1.4)-1.4)/1.4 = 114% slower than raw VIA applications and marshaling overhead would account for (12.6-11-0.1)/(12.6-11+1.4)=50% of latency, which translates into a 50% reduction in application bandwidth, compared with raw VIA. To make DCOM a competitive programming environment in the high-performance application market, we must apply optimizations at the runtime and marshaling layers to reduce overhead.

4. Millennium Falcon: Fast DCOM over VIA

Motivated by the overhead analysis described in the previous section, our Millennium Falcon prototype applies optimization at three layers of the DCOM architecture. At the transport layer, we take advantage of RPC semantics to perform efficient flow control. At the marshaling layer, we provide support for marshaling and unmarshaling of DCOM application buffers directly to and from network adapters. At the runtime layer, we analyze the critical paths of DCOM calls and build a new RPC binding-handle module to eliminate unnecessary RPC runtime overhead for DCOM applications. The effectiveness of the transport-layer optimization is specific to VIA, while the other two optimizations are applicable to the existing DCOM/RPC protocol stacks as well. We implemented the optimizations by modifying the NT4 SP3 DCOM and RPC source code and adding new modules for the new stack.

4.1. Transport and Marshaling Optimization for User-Level Networking

4.1.1. RPC Loadable Transport

To build a new RPC loadable transport for user-level networking, we first implemented a socket layer over VIA to hide the details of memory registration, VI management, etc. We then built the VIA loadable transport module on top of the socket layer to make it portable across different hardware and VIA implementations.

To preserve performance in a layered architecture, special attention must be paid to flow control. In VIA, incoming data is delivered directly, through DMA, to user-level buffers without kernel buffering. The receiver must post its buffers before the sender commences transfer; otherwise, VIA fails the transmission and closes the connection. For reliable communication, VIA applications must employ active flow control. Our first choice was to implement flow control in the socket layer. However, preliminary measurements showed that socket-layer flow control adds about 130 µs to the round-trip time, severely restricting performance. This overhead is mainly due to context switches and additional interrupts on both sides.

While socket-layer flow control is necessary for arbitrary applications, more efficient flow control can be achieved at a higher level by exploiting the RPC semantics. To achieve this, our socket layer supports optional disabling of flow control. Our loadable transport performs RPC-specific flow control as follows. When the server-side transport is about to accept a connection request from a client, it pre-posts a buffer for receiving the first method call so that the client can make calls immediately after it successfully makes a connection to the server. Before the client-side transport sends out the marshaled method call, it pre-posts a buffer for receiving the reply so that the server is free to send back a reply any time after the request is processed. Similarly, before the server sends a reply, it pre-posts a buffer for receiving the next request from the client. Essentially, in the RPC context, flow control messages are piggybacked on request and reply messages. Such an optimization is compatible with the original implementation in which concurrent calls between threads of the same pair of processes do not share the same socket connection. With this optimization, round-trip latency is reduced to 30 µs for small data sizes, an order-of-magnitude improvement over the transport overhead for DCOM over TCP (313 µs).

Two issues remain beyond the above simple flow control. First, VIA transmission will fail if the receiver buffer is smaller than the incoming data packet. In our current prototype, both the client and the server post buffers with size equal to the MTU of the network, which is close to 64KB on GigaNet GNN1000. A more practical solution is to adopt a default size smaller than the MTU and sufficient for most cases. When one side occasionally

needs to send data larger than the buffer size, it first sends a control message (with 30-µs round-trip time) to the other side to request a larger buffer, and then sends the actual data. Alternatively, if the network adapters support true Remote Direct Memory Access (RDMA) mode, the control message can request a large RDMA buffer on the receiving side and the actual data can be transferred using one VIA descriptor.

Second, even in the RPC context, there can be non-RPC-style communication. For example, as will be discussed later, Millennium Falcon uses Windows NT LanManager challenge-response protocol for connect-time authentication. This protocol consists of three legs: client sends an authentication request, server replies with a challenge, and finally client sends a response. At this point, the client is free to send its first request if using a traditional kernel-mode network protocol. In our prototype, special care needs to be taken because we rely on the RPC semantics for flow control. A straightforward solution is to add a fourth leg from the server back to the client to maintain the RPC semantics. Alternatively, the third leg can be combined with the first client call.

4.1.2. Eliminating Data Copying

While user-level networking eliminates data copy between user-level buffers and kernel buffers, the data marshaling process in DCOM and RPC introduces another data copy at a higher layer. Specifically, the proxy code is responsible for gathering method call parameters and packing them into an RPC buffer. This buffer can be directly accessed by the network adapters without additional copying. The analysis in Section 3 showed that the remaining data copy at the marshaling layer could significantly limit the achievable application bandwidth. Intuitively, because VIA supports scatter-gather operations and data conversion is not needed within a homogeneous cluster, it is possible to have network adapters directly access the memory regions of each individual call parameter without an intermediate copy. One could imagine a general scheme in which the IDL compiler is modified to generate proxy/stub code that does not copy call parameters into the RPC buffers; instead, the code constructs a gather or scatter list for VIA DMA access.

While such a general solution to optimal marshaling performance in user-level networking is beyond the scope of this paper, our current prototype provides support to reduce data copying on both the client and the server sides. Applications that require high bandwidth can supply modified proxy/stub code to take advantage of the support. We now describe our marshaling optimization by following the sequence of steps of a method call that involves only simple arrays of basic types.

Client side sending a request. A new RPC flag is added to allow modified proxy/stub code to declare that it is requesting the scatter-gather mode of transmission. When this flag value is set, the RPC runtime interprets the data buffer as a list of scatter-gather entries, each consisting of a starting memory address and a data length. The RPC runtime adjusts the entries to include RPC headers and passes the list down to the loadable transport, which calls the socket layer to perform a gather-send operation. Unflagged proxies and stubs still use copy-mode marshaling.

Server side receiving a request. Since the server-side RPC runtime may receive calls on any methods supported by the server process, it is in general not possible to specify a receive scatter list for any arbitrary method in advance. However, since the receiving RPC buffer is dedicated to the on-going RPC call for its entire duration, the stub code and the method implementation can use the buffer data directly without copying. Even for complex data structures, intelligent proxy/stub code can adjust data layout to use the incoming buffer directly [C98].

Server side sending a reply. This is basically similar to the client-side send. The only complication is when the client passes a pointer as an [out] parameter to request a variable-size array, whose size is determined at run time by the server. In this case, a properly designed COM interface method allocates space for the resulting array by calling *CoTaskMemAlloc()*, and the client is responsible for calling *CoTaskMemFree()* to free the buffer. To support this memory allocation paradigm, the original proxy/stub code does the following: the server-side stub frees the server-side application buffer once it has been copied into the reply RPC buffer; the client-side proxy allocates an application buffer once it knows the proper size, and copies the data from the RPC buffer to the application buffer.

On a user-mode network, the stub code cannot free the application buffer until the loadable transport has finished sending out the entire data directly out of the buffer. To simplify the task of modifying the standard proxy/stub code to delay freeing the buffer, we allow the stub to pass down a *callback function pointer* and a *context pointer*, along with the scatter-gather list. When the loadable transport finishes using the application buffer, it invokes the callback function with the context pointer. The callback function frees the application buffer.

Client side receiving a reply: Unlike the server side, since the client-side receive operation expects a reply for a particular method call, the proxy code can specify a reply scatter list to transfer incoming data directly into the memory of the [out] parameters and the return value. However, the variable-size array example mentioned previously poses certain restrictions. First, the size of the return array is unknown to the proxy when the call is made. It is not possible for the proxy to pre-allocate a buffer of the right size and post it in a scatter list. The reply must be received by an RPC buffer in this case. Furthermore, unlike the server side, it is not desirable for the client application to use the data directly from the RPC buffer because it may hold on to the buffer for an

undeterminable period before calling *CoTaskMemFree()*. So a memory copy is still needed in this case.

4.2. RPC Runtime Optimization for DCOM

Once the transport and marshaling optimizations are in place, the fixed runtime overhead of 100 µs becomes highly visible. In this section, we investigate how RPC runtime can be optimized to efficiently support DCOM. Optimizations at this layer are generally applicable to speeding up DCOM on any network, although the performance gain is much more significant in the SAN environment. As briefly introduced in Section 2, DCOM was designed as a thin layer on top of DCE RPC to leverage existing distributed system functionality provided by RPC. We re-examine this design decision from a performance point of view. Our study shows that, while DCOM benefits from the threading, connection, and security management support of RPC, it suffers from the unnecessary performance penalty due to RPC's interface management. We then describe the design and implementation of a new RPC runtime module that is optimized for DCOM.

4.2.1. Critical Path Analysis

Distributed object systems such as DCOM provide an abstraction to simplify distributed programming by hiding low-level communication details. To identify critical paths and apply effective optimization, we must understand how network traffic is generated in response to DCOM calls. The following is a critical-path analysis of critical events for typical DCOM client-server interactions.

Getting the first interface pointer to an object. When a DCOM client invokes *CoCreateInstanceEx()*, the client-side Service Control Manager (SCM) forwards the request to the SCM on the server machine. The server-side SCM locates or starts a process capable of hosting objects of the requested class. When the object implementation returns a pointer to the requested interface, DCOM marshals that pointer into an object reference by generating a 128-bit locally unique *Interface Pointer Identifier* (*IPID*) that identifies that particular interface instance. DCOM also registers with the RPC runtime the requested interface ID (IID) and an access control callback function if support for security is desired. Finally, the SCM asks the server to create an endpoint and construct a string binding. The SCMs then ferry both the object reference and the string binding back to the client process. Together the object reference and string binding uniquely identify the newly created interface instance. The client-side DCOM runtime constructs a binding handle from the string, loads the appropriate proxy code for the requested IID, and returns to the client a pointer to a proxy interface. Note that at this point no socket connection has been established between the client process and the server process.

We do not consider performance optimization for *CoCreateInstanceEx()* in this paper for two reasons. First, *CoCreateInstanceEx()* is only called once when the client initiates the first contact with an object. For applications that make a large number of subsequent method calls, object creation can be considered out of the critical path. Second, since *CoCreateInstanceEx()* involves expensive object creation logic, its performance will not be significantly improved by low-level runtime and transport optimizations alone. Changing the application architecture to move *CoCreateInstanceEx()* out of the critical path would be a much more effective solution. For example, server process *A* can pre-fetch and cache interface pointers from server process *B* running on another node of the same SAN cluster.

Making the first call to a server process. When the client makes the first call on a newly acquired interface pointer, the RPC runtime tries to reuse an existing socket connection to the server process that hosts the target object. If none is available, it opens a new socket connection. After the server accepts the request, if the authentication mode is turned on, the client starts the authentication process by sending a BIND message. Different authentication protocols may require different numbers of security related messages to be exchanged. As the result of a successful authentication, *security contexts* representing the calling principal are established on both sides. Finally, the client-side RPC runtime sends the request on the newly opened connection. Since the overhead of this entire connection process is amortized across all method calls between the client-server pair, it can be considered out of the critical path. If desirable, a process can pre-fetch an interface pointer from another process and make one initial call on it to set up the connection and security contexts in advance.

Making the first call to an interface. As part of the BIND process for establishing security contexts, a *presentation context* associated with the IID of the call is also established. A per-IID binding entry is inserted into the binding dictionary on both sides. The purpose of the presentation context is to ensure that the server indeed supports the requested IID, and to facilitate future call dispatching on the server side, which basically involves mapping an IID to the dispatch function and security callback function that were registered for it. When the DCOM client makes the first call to another IID of the same process, the RPC runtime sends an *alter-context* round trip to establish a presentation context for the new IID. In contrast, the socket connection and its associated security contexts are shared across IIDs.

Since it is quite common for DCOM clients to request an interface pointer and make only a very small number of calls on that pointer, the first calls to each IID should be considered inside the critical path. The extra alter-context round-trips may impose serious performance degradation. From DCOM's point of view, these round trips are needless overhead for three reasons. First, when a DCOM client successfully obtains an interface pointer through either the *QueryInterface()* or *CoCreateInstanceEx()* calls,

it has the implicit acknowledgement from the server that the IID is supported and has been properly registered. The additional RPC-level verification is redundant. Second, because all DCOM interfaces register the same RPC dispatch function, namely the entry point to the DCOM stub manager, it is unnecessary for the RPC runtime to lookup the binding dictionary to map an IID to its dispatch function. (The true DCOM call dispatching is performed inside the stub manager based on IPID.) Third, since all DCOM interfaces register the same per-process RPC access control function, the RPC runtime IID to security callback mapping is not needed.

Based upon the above analysis, we conclude that the interface (IID) management at the RPC runtime layer is unnecessary for DCOM applications. In our new binding-handle module optimized to run DCOM, we removed the entire IID notion from this layer to eliminate unnecessary network traffic, memory consumption, and performance overhead.

Making additional calls. The rest of the calls are certainly on the critical path. Removing the notion of IID from the RPC runtime layer improves the performance at several places along the critical path. On the client side, the binding dictionary lookup is eliminated. Since this operation required locking a dictionary shared by multiple threads, the performance gains are even larger for multi-threaded clients. The RPC runtime need not match presentation context to find a connection with the correct security context. On the wire, the IID is not transmitted because the IPID alone is sufficient for correct dispatching. On the server side, the RPC runtime invokes the security callback function and the DCOM stub manager directly without any dictionary or table lookup. Latency numbers in Section 5 demonstrate the effectiveness of these optimizations.

4.2.2. Binding-handle Optimization

We implemented our RPC runtime optimization as a new binding-handle module. This module sits under the common RPC API layer, which provides handle-type-independent processing. Our new module operates at the same level as the three existing binding-handle modules for connection-oriented protocols, connectionless protocols, and Local Procedure Calls. On the client side, when a call is made on a handle instance, the module finds the target-server endpoint. Then it searches for an available socket connection for the endpoint and, if there is none, creates a new connection. On the server side, the module maintains a thread pool based on the total number of active calls. Each thread, upon receiving a client request, performs appropriate security checks and forwards the request to the DCOM stub manager.

Security is one of the most important features provided by DCOM and RPC, and it plays an essential role in commercial client-server applications. DCOM security consists of three parts: *authentication* for verifying clients' identities and messages' authenticity; *access control* for restricting object access rights to a subset of users; and *impersonation* for allowing servers to execute under clients' credentials. Our new module supports all three security functions, while taking into account the special characteristics of the SAN environments.

DCE RPC supports multiple levels of authentication including *connect-time-only authentication*, *per-call or per-packet header protection*, *per-packet payload protection*, and *per-packet encryption*. Because our target environment is a physically secure server cluster, the last three protections against malicious attacks from outside agents are unnecessary. However, connect-time authentication is still necessary to prevent one legal user from gaining unauthorized access to the private data of another legal user. Specifically, once a client obtains an interface pointer, almost all future interactions between the client and the server will happen directly between the DCOM and RPC runtime libraries running inside the application processes. Therefore, even if the machines and kernels within the cluster trust each other, DCOM applications do not necessarily trust each other and so support for connect-time authentication is necessary. Fortunately, the overhead of this level of security is mostly outside the critical path, as discussed previously.

Our current prototype uses Windows NT LanManager challenge-response protocol for connect-time authentication. Once the client is authenticated, security contexts established on both sides serve as the basis for other security functions. When the server invokes DCOM APIs to impersonate the client, the task is mostly delegated to the security context. When the DCOM runtime performs an access permission check, it first impersonates the client and then retrieves the thread token to check against the access control list. Since DCOM provides APIs for client applications to dynamically change security-related information on a per-call basis, the client-side binding-handle module ensures that every method call is sent along a socket connection with the correct security context.

5. Performance Measurements

In this section, we present the performance comparison between our Millennium Falcon prototype and the existing DCOM implementation to quantify the performance gains of our optimizations. We use the same hardware setup as described in Section 3 for most of the measurements.

5.1. Round-Trip Latency

Figure 3 compares the round-trip latency for data sizes ranging from 0 to 8K bytes. The curves marked *VIA-copy* and *VIA-direct* represent DCOM over VIA with and without RPC buffer copying, respectively, as described in Section 4.1.2. For data of size zero, our implementation reduces the round-trip latency from 413 µs to 74 µs, a more than 5-time improvement. (The best number is 72 µs, with 74 µs being the average over 1,000 runs.)

Profiling data indicate that the overhead distribution is approximately 42 µs for runtime and 32 µs for transport, compared to 100 µs and 313 µs in the TCP case. For data of size 8K bytes, VIA-copy reduces the latency from 1540 µs to 457 µs and VIA-direct further reduces the number to 268 µs. The analytical equation for the VIA-direct curve is 22 µs (marshaling) + 42 µs (runtime) + (32 µs + 21 µs/KB * size) (transport). (The marshaling overhead as shown here is independent of the data size, but it can be a function of the number of call parameters in general.)

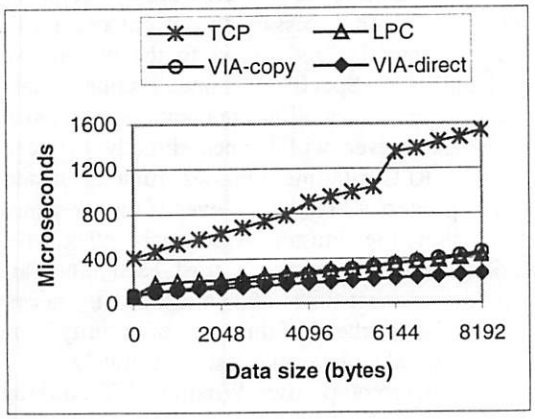


Figure 3. Comparison of DCOM round-trip latency (*PingPong*) over TCP, LPC, and VIA with and without data copying.

We also include the curve for DCOM over *Local Procedure Calls (LPCs)* in Figure 3. (LPC is one of the cross-process communication mechanisms within a single Windows NT machine.) The figure shows the interesting fact that, for this data range, cross-process local DCOM calls and cross-machine remote DCOM calls over VIA (with buffer copying) actually have very similar performance. The null-call round-trip latency for the DCOM over LPC case is 73 µs, where 45 µs comes from runtime and 28 µs from LPC. By eliminating data copying, the latency of VIA-direct at data size 8K is only about 60% of the corresponding DCOM-over-LPC latency. This result can have great impact on system designs that must decide between local versus remote execution and communication. For example, algorithms that tend to place communicating processes on the same machine in order to minimize communication overhead now have the flexibility of spreading processes across the network to take advantage of remote processing power.

For Figure 3, both client and server threads spin waiting for an incoming message to avoid delays due to context switching. For a discussion on the performance impact of blocking, see the full technical report [L98].

5.2. Application Bandwidth

Figure 4 compares DCOM application bandwidth for the three implementations. Due to the high overhead of the kernel-mode protocol stack, inefficient transport management assuming small network MTUs, and several intermediate buffer copies, the current DCOM-over-TCP implementation can only deliver a maximum bandwidth of 11.4 MB/s. Using user-level networking and taking advantage of the large MTU, VIA-copy increases the maximum bandwidth to 43.4 MB/s. However, that is still less than 50% of the raw VIA bandwidth of the GigaNet GNN1000 (about 105 MB/s). By eliminating all buffer copies on both sides, VIA-direct achieves a bandwidth of 86.1 MB/s.

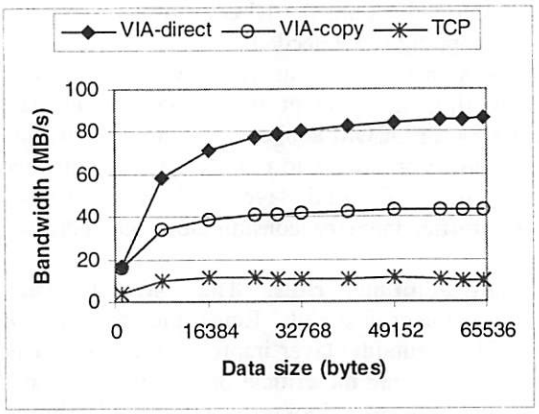


Figure 4. Comparison of DCOM application bandwidth (*PingPong*) over TCP and VIA.

The dramatic increase in available bandwidth demonstrates the importance of the marshaling layer optimization. Today, many applications use DCOM as their main programming paradigm, but use separate, lower-level communication primitives such as sockets for bulk data transfer. Since the VIA-direct implementation preserves 96% of the bandwidth available to a raw socket implementation (90.1 MB/s), a consistent programming paradigm based on DCOM can now be applied throughout an application for both control and bulk data transfer.

5.3. Secure Calls and First Interface Calls

Next, we compare the round-trip latency of secure calls and the first call to a new IID (after a socket connection has been established). The top portion of Table 2 shows that, with security turned off, the latency of the first call to an IID is more than twice that of later calls in the TCP case. Overhead comes from the alter-context round trip (Section 4.2.1), the initialization of the RPC binding handle, and the initialization of the DCOM channel object. By eliminating the alter-context round trip and by moving the initialization of RPC binding handle out of the critical path, our implementation achieves a round-trip latency of 109 µs, which is 8 times faster than its TCP counterpart. The remaining overhead of 109-74=35 µs is due to per-interface DCOM channel initialization for supporting per-interface security.

The lower portion of Table 2 shows latency when connect-time authentication is enforced. Ideally, this level of security should impose zero overhead for all calls using

an existing connection. However, the current DCOM-over-TCP implementation has a 448-413 = 35 µs overhead that comes from two sources. First, the client-side RPC runtime must match, on a per-call basis, the security setting of the current call against the security context of existing connections. The second source of overhead is an implementation issue where the RPC layer performs security tasks that are unnecessary for DCOM. By removing the second source in our binding-handle module, the per-call overhead is reduced to only 76-74 = 2 µs.

Without Security	**Later Calls**	**First Calls**
DCOM over TCP	413 µs	880 µs
DCOM over VIA	74 µs	109 µs
With Security (Connect-time-only)	**Later Calls**	**First Calls**
DCOM over TCP	448 µs	1600 µs
DCOM over VIA	76 µs	300 µs

Table 2. Comparison of DCOM round-trip latency for secure and non-secure, first and subsequent interface calls.

The lower portion of Table 2 shows that, for DCOM over TCP, the latency of first secure calls is more than 3 times that of later calls. A detailed discussion on the source of the overhead and our optimizations can be found in [L98]. For the VIA case, out of the 300 µs, about 186 µs are spent on acquiring the caller's credentials for supporting dynamic security.

5.4. Apartment Threading

DCOM supports two different threading models for servers: free threading and apartment threading. So far, we have limited our discussion to the free threading model, in which the RPC threads responsible for receiving client requests are also the threads that eventually invoke the object stubs. In the free threading model, the application programmer must synchronize data accesses. In contrast, apartment threading simplifies concurrent programming by providing automatic access synchronization. By having a single thread operate in a Single-Threaded Apartment (STA), all accesses to objects in that STA are serialized and there is no need for additional programmatic synchronization.

The servers-side DCOM runtime implements apartment threading by posting each incoming DCOM call to the Windows message queue associated with the target STA, thereby serializing all calls into that STA. Apartment-threaded servers are in general slower than free-threaded servers due to the context switches between RPC and application threads. Unlike transports based on custom marshaling, Millennium Falcon preserves full support for apartment threading.

Our measurements show that, for both TCP and VIA, apartment threading adds an additional 90 to 160 µs on top of the free-threading numbers across different data sizes. (See [L98] for performance graphs.) In particular, the null-call latency in the VIA case more than doubles to 168 µs, and the maximum bandwidth is reduced by 7% to 80.3 MB/s. This suggests that the current implementation of apartment threading may need to be redesigned to take full advantage of user-level networking.

5.5. Microsoft Transaction Server

Microsoft Transaction Server (MTS) provides a runtime environment combining the features of a Transaction Processing (TP) monitor and an object request broker. MTS supports the notion of automatic transactions: a developer builds a COM object in DLL form and registers it with MTS. When a client activates the object, an MTS surrogate process loads the DLL and automatically wraps it with a transaction context so that the object can participate in transactional interactions. MTS is currently built on top of apartment threading.

Table 3 compares the latency for null MTS calls. For the VIA case, of the 280 µs for an MTS call with role-based security, 168 µs are from apartment threading, 6 µs from transactional wrapping, 2 µs from DCOM security, and 104 µs from MTS security. More efficient implementations of apartment threading and MTS security are needed once the DCOM and RPC layers have been optimized for user-level networking.

	Without Security	With MTS Security
MTS over TCP	558 µs	698 µs
MTS over VIA	174 µs	280 µs

Table 3: Round-trip latency comparison for MTS calls.

5.6. Real Applications

We next discuss two categories of DCOM applications and present some preliminary results. The first category is *off-the-shelf consumer applications*. We use the distributed version of Microsoft *PhotoDraw 2000*, created by the Coign auto-partitioning tool [H99] to run on two machines. *PhotoDraw 2000* is an application for manipulating digital images. It is composed of approximately 112 COM component classes in 1.8 million lines of C++ source code. In the particular runs used in our measurements, *PhotoDraw 2000* loaded a 3MByte graphical composition from storage, displayed the image, and exited. It created 295 COM objects and made approximately 9000 DCOM calls. Results showed that Millennium Falcon reduced the total execution time from 24.0 seconds to 21.8 seconds, a 9.2% improvement in performance. On average, the per-call performance gain is around (24.0 - 21.8) / 9000 = 244 µs.

The second category is *client/server applications involving database operations*. Our measurement shows that an MTS-over-TCP call involving an SQL SELECT operation takes 3.45 ms on SQL Server 6.5. Making the same call over Millennium Falcon reduces the round-trip latency to 3.14 ms, a 9.0% performance gain. For an SQL

INSERT operation, the corresponding numbers are 5.06ms vs. 4.79ms, a 5.3% improvement.

6. Related Work

Madukkarumukumana *et al.* built a custom marshaling layer for DCOM over VIA [Ma98]. Their implementation sacrifices DCOM features for speed. Zimmer and Chien built a UDP loadable transport for MSRPC over Illinois Fast Messages [Z98]. They pointed out that the current RPC implementation imposes serious limitations on potential performance gains. Our work on RPC runtime optimization was partly motivated by their observations.

Bilas and Felten [B97] modified SunRPC to run over Shrimp VMMC. In their SunRPC-compatible implementation, their marshaling layer optimization and ours share similar basic ideas, but differ in IDL semantics. Their second system, ShrimpRPC, forgoes application compatibility with SunRPC to allow further optimizations. In contrast, our goal was an efficient RPC layer capable of supporting existing DCOM applications.

Gokhale and Schmidt [G96][G97] evaluated commercial CORBA and RPC implementations. They discovered that some had inefficient server-side dispatching, and that the conversion of typed data to XDR format in SunRPC is a major source of extraneous overhead for homogenous machines. Unlike the studied CORBA implementations, DCOM's dispatch mechanism is already highly optimized, and the NDR conversion in MSRPC occurs only at the receiver side and only if sender and receiver employ different representation formats.

Two classes of optimization have been proposed for the marshaling layer. The first class reduces data copying. Thekkath and Levy [T93] marshaled RPC arguments directly in the kernel to avoid data copying to kernel buffers for ATM and FDDI. This optimization is unnecessary for user-level networking. Some of our optimizations are similar to the buffer caching and aggregation used in [D93], although they dealt with additional cross-domain issues. The second class of marshaling optimizations applies known compiler transformations to stub generation [E97][Mu98]. Such optimizations are essentially orthogonal to our work.

7. Conclusions

We have demonstrated an effective approach to reducing round-trip latencies and increasing application bandwidth for a commercial distributed-object system over user-level networking. Just as high-speed networks shifted the performance bottleneck to the protocol stacks and user-level networking shifted the bottleneck to the communication infrastructures of distributed object systems, our optimizations have again shifted the bottleneck to the support for security and threading, and the initialization overhead of internal data structures. Performance measurements suggest that existing architectures and implementations in these areas need to be carefully reevaluated in order to take full advantage of high-speed networking.

Acknowledgement

We thank Jim Gray, Karin Petersen, and the anonymous reviewers for their valuable comments.

References

[B97] A. Bilas and E. W. Felten, "Fast RPC on the SHRIMP Virtual Memory Mapped Network Interface," in *J. Parallel and Distributed Computing*, Feb. 1997.

[B94] M. A. Blumrich et al., "Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer," in *Proc. Int. Symp. on Computer Architecture,* pp. 142-153, 1994.

[B98] N. Brown and C. Kindel, Distributed Component Object Model Protocol -- DCOM/1.0, 1998.

[C95] The Common Object Request Broker: Architecture and Specification, Revision 2.0, July 1995.

[C98] C.-C. Chang and T. von Eicken, "A Software Architecture for Zero-Copy RPC in Java," Cornell CS Technical Report 98-1708, Sep. 1998.

[D95] DCE 1.1: Remote Procedure Call Specification.

[D93] P. Druschel and L. L. Peterson, "Fbufs: A High-bandwidth Cross-domain Transfer Facility," in *Proc. SOSP, 1993.*

[E97] E. Eide et al., "Flick: A flexible, optimizing IDL compiler," in *Proc. ACM SIGPLAN'97 Conf. On Programming Language Design and Implementation (PLDI),* pp. 44-56, June 1997.

[G96] A. Gokhale and D. C. Schmidt, "Measuring the Performance of Communication Middleware on High-speed Networks," in *Proc. SIGCOMM*, Aug. 1996.

[G97] A. Gokhale and D. C. Schmidt, "Measuring and Optimizing CORBA Latency and Scalability Over High-speed Networks," *IEEE Trans. on Computers*, Vol. 47, No. 4, 1998.

[H99] G. Hunt and M. L. Scott, "The Coign Automatic Distributed Partitioning System," in *Proc. OSDI,* 1999.

[L98] L. Li et al., "High-Performance Distributed Objects over a System Area Network," Tech. Rep. MSR-TR-98-68, Microsoft Research, 1998.

[Ma98] R. S. Madukkarumukumana, C. Pu, and H. V. Shah, "Harnessing User-Level Networking Architectures for Distributed Object Computing over High-Speed Networks," in *Proc. USENIX NT Symposium,* pp. 127-135, Aug. 1998.

[Mu98] G. Muller et al., "Fast optimized Sun RPC using automatic program specialization," *Proc. ICDCS, May 1998.*

[P97] S. Pakin, V. Karamcheti, and A. A. Chien, "Fast Messages: Efficient, Portable Communication for Workstation Clusters and MPPs," *IEEE Concurrency,* 5(2):60-73, 1997.

[T93] C. A. Thekkath and H. M. Levy, "Limits to Low-latency Communication on High-speed Networks," *ACM Trans. on Computer Systems,* 11(2):179-203, 1993.

[V97] Virtual Interface Architecture Specification, Version 1.0, Dec. 1997. (http://www.viarch.org)

[V95] T. von Eicken, A. Basu, V. Buch, and W. Vogels, "U-Net: A User-Level Network Interface for Parallel and Distributed Computing," in *Proc. ACM SOSP, 1995.*

[W95] A. Wollrath, R. Riggs and J. Waldo, "A Distributed Object Model for the Java System," *USENIX Journal, Computing Systems*, Vol. 9, No. 4, pp.265-289, 1996.

[Z98] O. M. Zimmer and A. A. Chien, "The Impact of Inexpensive Communication on a Commercial RPC System," submitted, 1998.

MTEX - A Bridge For Migrating CAD Design Environment From UNIX To NT

Ty Tang, Vipul Lal, Shesha Krishnapura
ty.tang@intel.com, vipul.lal@intel.com, shesha.krishnapura@intel.com
Design Technology, Intel Corporation

Abstract

This paper shares an innovative technology that we developed while migrating Intel CAD design environment from UNIX to Microsoft Windows NT operating system. The UNIX to Windows NT migration is a complex and challenging task because the chip design environment involves CAD applications, CAD infrastructure scripts, design flows made up of UNIX-centric scripts and design data.

Due to the wide differences between UNIX and NT scripting architectures, straight porting to native NT scripting environment is not feasible without a complete redesign and redevelopment of the CAD infrastructure and design flow scripts. Our approach to this problem was to develop a transition production-capable mixed NT-UNIX CAD environment technology, MTEX, with the eventual goal of complete migration to a Windows NT CAD environment. This technology solves the script migration problem and supports a seamless mix of UNIX and NT centric CAD tools.

In this paper we will present MTEX functionality, its internal design and architecture.

1. Introduction

Intel's current CAD environment is based on high-end UNIX based RISC workstations. To harness the power of the cheaper Intel Architecture (referred to as IA from here on) platform and to have a single workstation that supports both engineering and office tools, Microsoft Windows NT (referred to as NT from here on) was chosen as the operating system of choice. The UNIX to NT migration tasks were broken down into the following three activities:

- Porting of CAD tools
- Porting of the associated runtime environment
- Porting of the test environment

Most of the CAD tools are written in high-level language such as C or C++, whereas the runtime and the test environment were mostly UNIX centric C-shell and Perl scripts.

Initially we estimated, rather incorrectly, that most of our effort would be used in porting the design tools to native Win32 APIs. As we progressed further, it became evident that the porting of C and C++ tools to Win32 APIs does not constitute a big task.

Porting of runtime and test environment to Windows NT became a bigger challenge due to the incompatibility of the scripting environment between NT and UNIX. The current commercially and publicly available UNIX utilities on NT were not mature enough to support a production-worthy UNIX-like scripting engine in multi-user mode that provides the identical UNIX functionality. Other possibilities were evaluated that included going in for one of the native Win32 scripting environment, DOS batch files, which was very limited in functionality or Visual Basic which will require complete re-architect of Intel CAD runtime and test environment.

We finally realized that complete migration in one-step to IA-NT as a CAD platform was not possible and a creative but simple technique was needed to meet our aggressive migration goal. This paper describes the problems in detail and our innovative technology to over come to the migration problems.

2. The Problem

The main problems that prevent Intel CAD design environment from moving directly to a pure IA-NT compute environment:

- UNIX-centric runtime and test environment scripts can not be reliably ported to IA-NT
- Not having the complete set of design CAD tools on IA-NT (compile time and run time dependent tools)

Intel CAD runtime and test environment is made up of millions of lines of UNIX-centric Perl and shell scripts. The runtime environment drives the microprocessor design flow process. These infrastructure scripts act as

gluing utilities that integrates the various CAD applications into a functional design environment. Some of the tasks include data format translation, data extraction, simulation, data analysis, waveform analysis, performance analysis, design validation, etc. The CAD runtime environment needs to be both accurate and reliable to support design processes that might run for days.

The test system for CAD tool suite consists of UNIX scripts which provide a dynamic and open environment in the sense that these scripts can take any number and type of test cases and analyzers as arguments. The test system scripts rely heavily on UNIX-centric process and user environment. Some tasks that appear straightforward on UNIX might turn out to be non-trivial on Windows NT. The following example illustrates the point.

Script 1: ***foo***

```
#!/usr/local/bin/perl
system("fool 'grep -i -v fail' datafile");
```

Script 2: ***fool***

```
#!/usr/local/bin/tcsh -f
cat $argv[2] | $argv[1]
```

ASCII file: ***datafile***

```
test data - success
test data - fail
```

Example 1

On UNIX, executing *foo* will output:

test data - success

On Windows NT, however, executing the same script will fail due to couple of errors (even under tcsh with proper TCSHSUBSTHB environment setup).

The first error happens when the script *foo* calls *fool*. Since *fool* is a tcsh script and Windows NT does not support execute permission on scripts, "tcsh -f" needs to be added to system() to invoke *fool* as follows:

system("tcsh -f fool 'grep -i -v fail' datafile");

The second error is due to the fact that Win32 Perl does not escape the single quote (') character. As a result, the second argument to *fool* on NT will be:

'grep -i -v fail'

instead of:

grep -i -v fail

as in UNIX. Of course, changing the single quote to double quote (") will get the script to work under NT.

In brief, we want to bring up a point that even though the commercial and public UNIX-like utilities are available on Windows NT, the native NT scripting environment does not remotely resemble UNIX scripting environment. During the porting of our CAD runtime and test system, we realized that the currently available UNIX utilities for NT can not support and maintain a production-level quality UNIX-like infrastructure environment for Intel CAD tool development and design activities on Windows NT.

Besides the many concerns we have with the UNIX-like tools' reliability and functional incompatibility on Windows NT, the main discouragement comes from the high failure rate of UNIX-centric scripts performing in the integrated UNIX-like scripting environment under Windows NT. We have pointed out some of the problems in example one above. The many "little" differences between UNIX and Windows NT such as escape characters, executable loading mechanism, search-path support for scripts, limited DOS shell, etc. make the runtime environment highly unstable, especially in cases where executable flows are moving back and forth between Perl and shell scripts. These scripts rely heavily on idiosyncratic UNIX environment to behave properly.

Example: *system($argv[1])*, where *$argv[1]* can be an executable, a Perl script, a shell script, or a command series such as *'awk ... | grep ... | head ...'*.

We came to the conclusion that we might be able to put a patch here and there to fix the immediate problems as they occur in our runtime and test environment. However, with millions of lines of scripts, it is very unlikely that we can achieve the stability level we need for our CAD development and design activities. We realized that we must not pretend that we are still using UNIX on NT and began to investigate alternatives to our problems.

3. The Analysis

Our project goal is clear and simple that is to move Intel CAD design environment to IA-NT. Two primary boundary conditions while meeting this goal are:

1. The skill set of Intel CAD tool developers and chip design engineers is UNIX-centric. Any chosen solution to migrate Intel CAD environment from UNIX to Windows NT needs to be least disruptive to these developers and design engineers.
2. Intel CAD environment consist of both internally developed and external CAD applications and libraries. The chosen solution must be in accordance with the computing standard of the EDA industry to assure the co-existence of internal tools with external tool libraries.

The right solution is to re-architect Intel CAD tools, runtime and validation environment to take advantage of native Windows NT features (Windows NT Logo standards, ActiveX controls, DCOM, etc.) coupled with Windows NT centric extension language interface to achieve the complete benefits of native Windows NT environment. However, this is undoubtedly a long-term project and we are working towards that direction.

We realized that the practical short-term solution is to use Windows NT CAD applications with pre-existing UNIX centric Intel CAD runtime and test environment scripts to enable Intel developers and design engineers a gradual exposure and a smooth migration to NT.

Our investigation of using exiting Win32 emulated UNIX scripting environment can not meet our need as we have illustrated in the earlier problem section. The second approach in trying to use the POSIX subsystem (both native Windows NT version and third-party version) also did not solve our problem because Intel CAD tools need to be native to Win32 subsystem due to external vendor libraries dependency. The POSIX subsystem does not work well in the areas of multi-level inter-process invocation between WIN32 applications and POSIX utilities, symbolic links for project data sharing, automatic environment variables conversion, and NT-UNIX shared file-systems support.

After several attempts to get a complicated UNIX scripting environment to work flawlessly and reliably on Windows NT, we re-examined our approach from a different angle. Instead of moving the UNIX centric CAD design environment to Windows NT, why not integrate native Windows NT applications into our pre-existing UNIX CAD environment?

With this refreshing idea, we set out to develop an in-house product, the MTEX technology (Multi-platform Tool Execution eXtensions). MTEX allows CAD tools to be configured to run on either UNIX or Windows NT workstations while the bulk of CAD runtime and test environment scripts remain on UNIX. This transition technology serves as a bridge to allow the gradual migration of Intel CAD tools and environment and the gradual update of Intel engineers' mind-set and skill-set from UNIX to Windows NT.

4. Design Goal

As a UNIX to Windows NT transition technology, the MTEX has a distinguished design goal:

Support a mixed NT-UNIX CAD design environment which enable gradual migration of Intel CAD design tools and design environment from UNIX to Windows NT. This requirement has two advantages:

a. Allows to selectively migrate the high compute usage CAD design tools from UNIX to native Windows NT to start taking advantage of the cheaper IA-NT computing cycles early.
b. Allows CAD developers to use the existing UNIX-centric test systems and test vectors to validate their design tools on Windows NT. This eliminates the uncertainty with the correctness of the newly developed test cases on Windows NT.

With time Intel's CAD design environment will be moving from UNIX majority to Windows NT majority design tools and scripts.

5. MTEX Overview

MTEX is a transition technology to enable the integration of a mixed NT-UNIX design environment. The idea behind the MTEX capability is to extend the remote procedure call concept to a remote execution environment. The result is a tool environment that allows transparent execution of CAD tools in mixed NT-UNIX platforms. In this cross-platform environment, the user will be working on an NT desktop and executing design commands from a remote UNIX xterm window using the same familiar design flow written in scripts. The user view is illustrated in Figure 1.

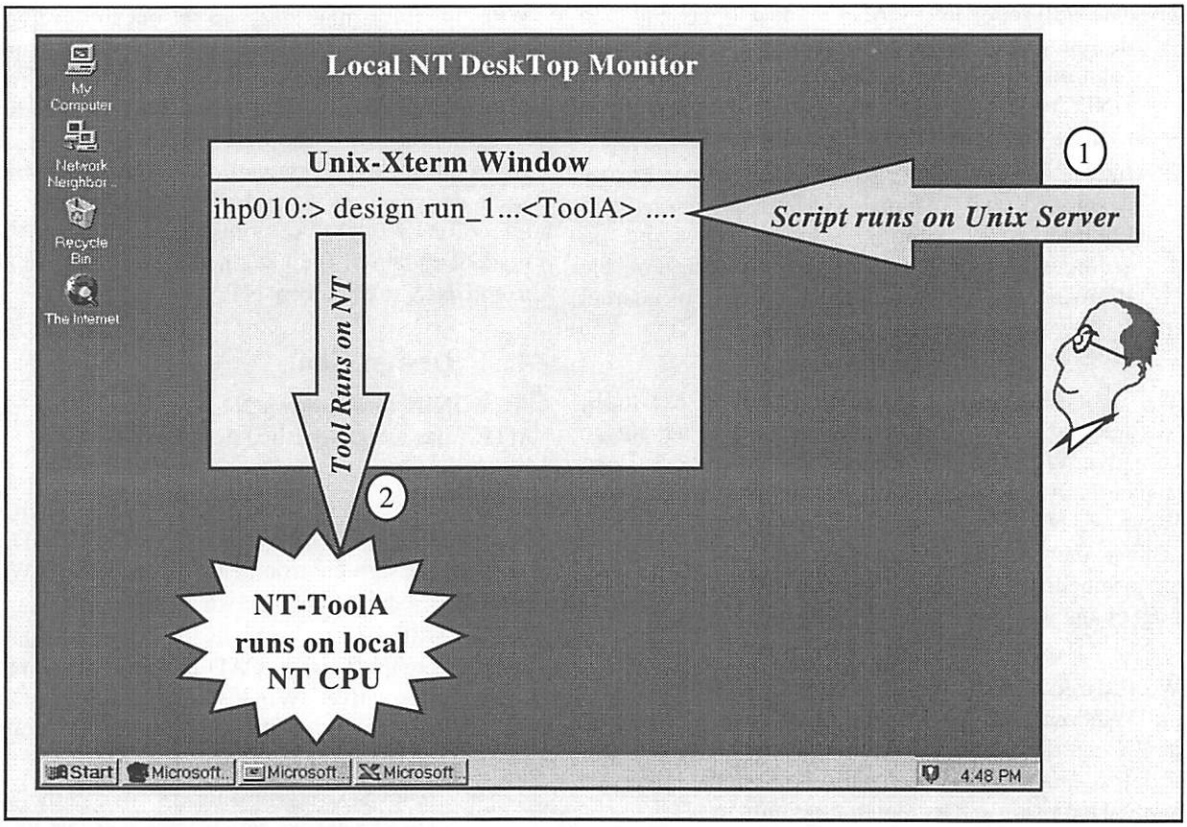


Figure 1: User View of MTEX Environment

6. MTEX Architecture

The following block diagram provides a high level architecture of the MTEX application.

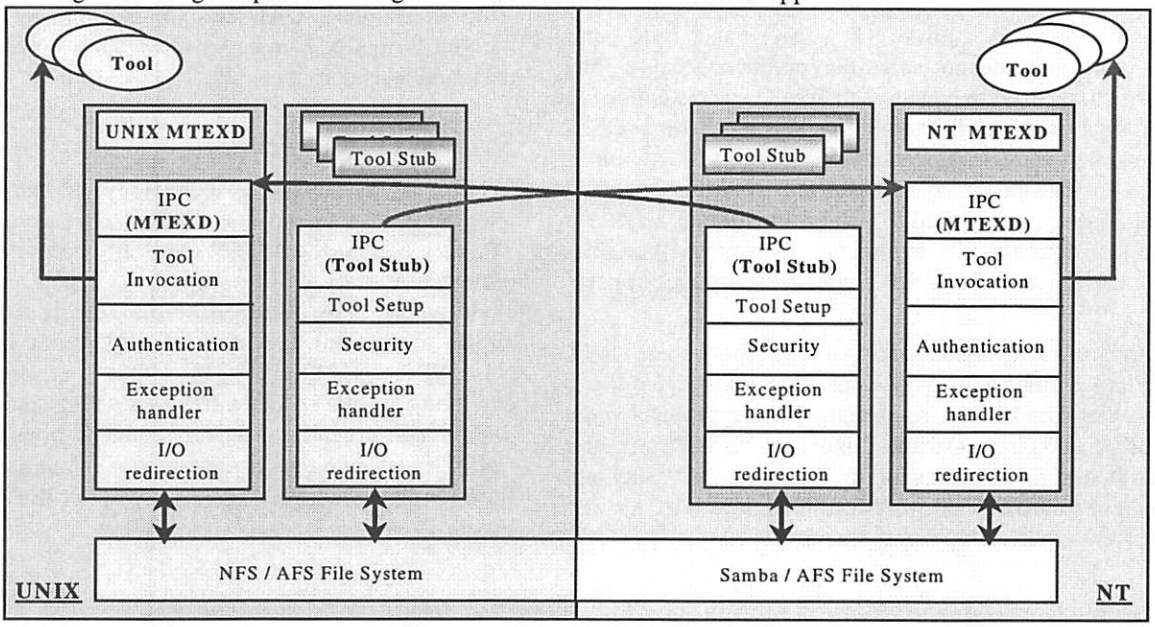


Figure 2: MTEX Architecture

6.1. MTEX Components

MTEX is implemented with three components: a UNIX server daemon, an NT server service, and CAD tools and scripts that are set up to run under MTEX.

Every tool executable in this environment has two complementary parts, namely the tool executable itself and the tool stub, each of which is running on different platforms. The tool stub is a wrapper running on local system that launches the tool executable on the remote system.

When a MTEX tool stub is executed on the local system, it takes a snapshot of the local run-time environment and sends this information to the server daemon (or service). The MTEX server on the remote system impersonates the local user, duplicates the local run-time environment with necessary platform specific adjustments, and invokes the actual tool executable. The MTEX server also handles the routing of STDIO streams to the tool stub and passes the exit code of the tool executable to the tool stub. The tool stub exits with the same exit code. An overview of how MTEX components work together is shown in Figure 3

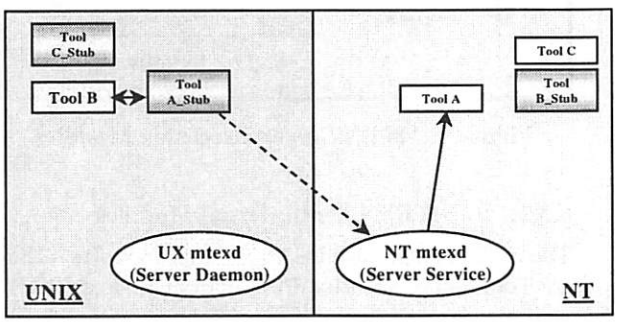


Figure 3: Overview of MTEX Components

6.2. MTEX Features

MTEX provides the following features to enable transparent remote tools execution:

1. Conversion and customization of the run-time tool environment. The run-time environment on the local system is duplicated on the remote system for tool execution.
2. NT and UNIX shared file-system support so that it can handle the tool stubs, executables, data, and output files residing on shared file systems.
3. Handling the redirection of standard input, standard output, and standard error streams between the tool stub on the local system and the tool executable on the remote system.
4. Providing the capability to execute binary files on the remote system (UNIX or NT).
5. Providing the capability to launch the scripts on the remote system (UNIX or NT).
6. Duplication of the shared file system credentials for transparent user access from local to remote system.
7. Impersonation of local system user on the remote system and execution of the tool with the same user credential.

In Figure 4 we illustrate how MTEX can be integrated into pre-existing UNIX CAD design environment. In the figure, a shared complementary directory structure is set up between UNIX and NT. The execution flow begins when the script *driver_script* invokes the tool named *testtool* on UNIX. Testtool is a MTEX client, which invokes the real testtool.exe on Windows NT. After *testtool.exe* has finished execution, the execution control returns to testtool stub on UNIX-end which then exits with testtool.exe's exit status. At this point, the *driver_script* continues with the execution flow to invoke the *analysis_script*, on UNIX.

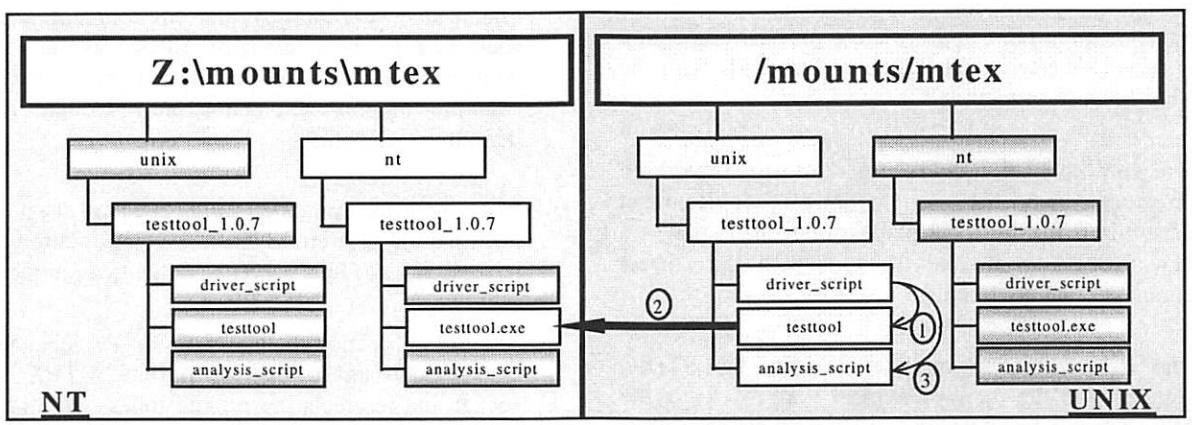


Figure 4: Complementary Directory Structure and Execution Flow under MTEX

6.3. MTEX Internals

MTEX features are implemented across the MTEX server and client. This section shows how the features work and how they are structured in MTEX server and MTEX client to provide a platform independent execution environment.

6.3.1. MTEX Server Internal Modules

As shown in Figure 5, the MTEX server has four major modules. They are IPC, Authentication, Tool Invocation, and I/O Redirection. For clarity purpose, we will omit the error and exception handling routines in this section and in the diagram.

MTEXD, the MTEX server, is to be started by a superuser on UNIX and a user with administrative privileges on NT. It can also be setup to be invoked as part of the boot-time initialization. The MTEX server listens for connection requests by MTEX clients. On detection of a connection request, the server starts a new thread of execution (or a new server process) to handle this request. The original thread returns to listen for client requests.

The IPC module receives the message packets from MTEX client (stub). It decodes the message and to retrieve the following important information:

- User's login information
- AFS token information
- Current working directory
- Customized runtime environment
- Application to be invoked and its arguments.

The Authentication module takes user login and AFS token information and authenticates with local operating system and the AFS server. If the authentication fails, appropriate error handling action is taken; otherwise, the execution proceeds with the Tool Invocation module.

The Tool Invocation module sets the current working directory and the tool runtime environment customized for the local platform. It then spawns a new process to invoke the tool with the correct command line arguments.

The I/O redirection module redirects the STDIO streams for the newly invoked tool back to the MTEX stub's STDOUT/STDERR sockets as well as redirects the stub's input data to the tool's STDIN stream. When the tool exits, the MTEX server gets the exit code of the tool and sends it back to the stub.

The execution flow of the above four MTEX server modules is illustrated in Figure 5

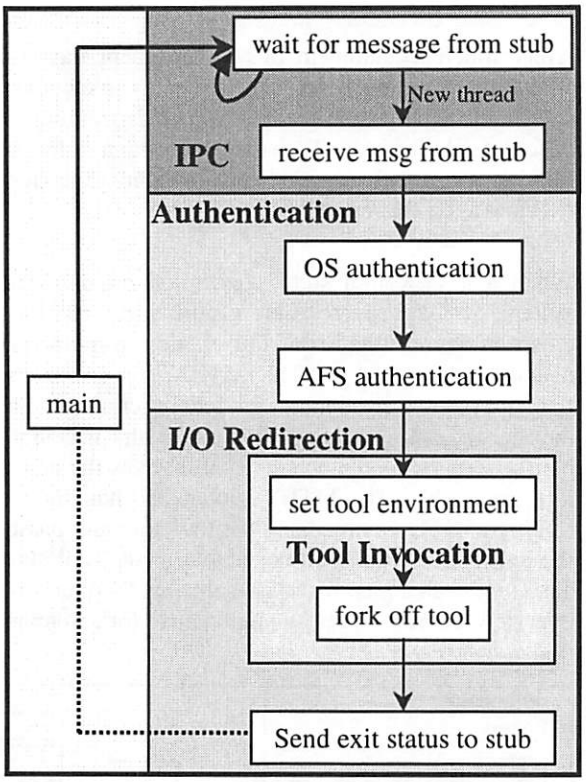


Figure 5: MTEX Server Execution Modules

6.3.2. MTEX Client Internal Modules

The MTEX client also has four major modules. They are Tool Setup, Security, I/O Redirection, and IPC.

The Tool Setup module collects user environment related data and converts the environment variables according to the conversion rules specified in the configuration file for the target platform. An example of this conversion may include adding default search paths.

The Security module collects the user login information as well as the AFS information to be processed by MTEX server's Authentication module.

The I/O redirection module redirects the local MTEX stub input data to the appropriate MTEX server socket and redirects the remote tool's output data to

the matching local stub's STDIO streams. The I/O redirection is totally transparent to the user.

The IPC module encodes the data collected by the Tool Setup and Security modules into message packets and sends them to the MTEX server. It then goes into a send/receive state until the lifetime of the tool. When it receives the exit message from MTEX server, it puts the stub to exit with the same status as that of the tool on the remote system.

The execution flow of the above four MTEX client modules is illustrated in Figure 6

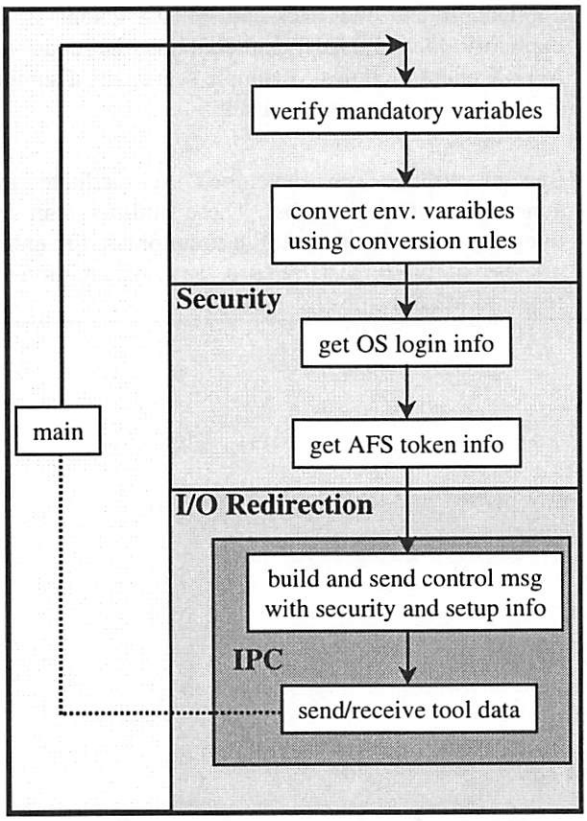


Figure 6: MTEX Client Execution Modules

6.4. MTEX Implementation

MTEX is developed for HP-UX 10.01, HP-UX 10.20, AIX 4.1 and Windows NT 4.0. The MTEX server has been implemented as a daemon on UNIX and as a Win32 service on Windows NT.

The IPC communication between MTEX server and client is over TCP using BSD Sockets on UNIX and Winsock 2.0 on Windows NT. AFS library calls are used by the Security module to pass in user's AFS tokens to MTEX server and by the Authentication module to establish user's AFS credentials on the server end. User authentication on Windows NT is achieved by using the Win32 API CreateProcessAsUser() to impersonate the current logged-on user. This has the implication that the user has to be logged-on to a Windows NT desktop in order to use that system as the remote system under MTEX.

The Tool Setup module copies relevant current user runtime environment such as working directory, process environment block, etc. while the Tool Invocation module recreates the user runtime environment on MTEX server end. Appropriate adjustments are made to the environment to account for syntax differences, system variables and path differences, etc. between the local and remote system as shown in the following example.

Original list of variables

```
DISPLAY=sccia677.sc.intel.com:0.0
HOME=/home/vlal
HOSTTYPE=hp9000s700
MTEX=/mtex_cad/hp700_ux100/mtex/v1.1
PATH=/bin:/usr/bin:/usr/vue/bin:/usr
/bin/X11:/usr/ccs/bin:.
```

Modified list of variables

```
DISPLAY=sccia677.sc.intel.com:0.0
HOSTTYPE=nt_4.0
MTEX=/mtex_cad/nt_4.0/mtex/v1.1
PATH=C:/WINNT;C:/WINNT/system32;X:/c
ygnus-b18/h-i386-cygwin32/bin;
X:/Perl/5.004_04/bin;.
PERL5LIB=X:/Perl/5.004_04/lib
```

Example 2: Environment Variables Modification

The variables can be divided into the following 4 categories based on the operations performed on them.

- Variables that remain unchanged (e.g. DISPLAY)
- Variables that are modified for the target platform (e.g. HOSTTYPE, MTEX,PATH)
- New variables that are introduced for the target platform(e.g. PERL5LIB)
- Variables that are blocked from being sent over to the target platform (e.g. HOME)

Besides the major modules, MTEX routines also support error and exception handling and recovery,

remote execution configuration and customization (e.g. MTEX_PREPATH=xxx will prepend xxx to the $PATH environment variable on remote machine), etc. To make the technology simple and easy to understand, we only cover the main features in this paper. All MTEX modules are implemented with documented library calls, system calls, and Win32 APIs.

```
#!/usr/intel/bin/perl

($scriptName = $0) =~ s/.*\//g;

# ...< check for required environment variables > ...

exec("$mtexDir/mtexstub", <Remote system tool path>, $scriptName, "@ARGV");

die "$scriptName -E- exec failed...";
```

Example 3: A simple UNIX front-end tool stub

A MTEX tool stub consists of two parts: the front-end stub and a back-end generic MTEX client. The front-end stub gets the name of its complementary tool on the remote system (e.g. the stub for foo.exe is foo).

On UNIX, the front-end stub is a Perl script that simply invokes the generic MTEX stub with all of its command line arguments. Example 3 presents a sample UNIX front-end tool stub.

The back-end generic MTEX client, known as mtexstub in the above example, is a thin client executable on UNIX. It implements the MTEX client modules described in section 6.3.2.

On Windows NT, the front-end stub is a simple 'C' application and the back-end MTEX client is an explicitly loaded DLL that contains the code for MTEX client modules. Example 4 presents a sample Windows NT front-end tool stub.

Special utilities are developed to facilitate the generation of tool stubs. These utilities can be integrated into a tool makefile to automate the entire process of build, test, release, and the creation of required MTEX stubs for the tool.

```
#include <windows.h>
#include <stdio.h>
int main(int argc, char **argv)
{
    int retCode = 1;
    HINSTANCE hMtexLib;
    Char buffer[260];
    FARPROC mtexEntry;

    Buffer[0] = \\0';
      .
      .
  // Get the MTEX environment variable. to load the mtexstub DLL
 sprintf(buffer, "%s/mtexstub.dll", getenv("MTEX"));
  // Load the MTEX stub library
  //
   if ((hMtexLib = LoadLibrary(buffer)) == NULL) {
      printf("%s: -E- Failed to load the library %s - %d\\n", argv[0], buffer, GetLastError());
      exit(GetLastError());
   }
  // Get the address of startMTEX function
  //
   if ((mtexEntry = GetProcAddress(hMtexLib, "startMTEX")) == NULL) {
     printf("%s: -E- Failed to get the entry point of the stub startMTEX -  %d\\n", argv[0], GetLastError());
     FreeLibrary(hMtexLib);
     exit(GetLastError());
```

```
    }

    // Invoke the startMTEX function with the requisite arguments
    // Expand the toolPath is it starts with a \$' - env. Variable
    //
    retCode = (\* mtexEntry)(argc, argv,<toolPath.....>);

  // Free the MTEX library
   FreeLibrary(hMtexLib);

    // Exit with the exit code returned by startMTEX(...) function
    exit(retCode);
}
```

Example 4: A simple MTEX NT front-end tool stub

7. User Experience

Over the past year, the CAD organization at Intel has successfully used MTEX to validate more than 40 IA-NT ported CAD tools (3 million lines of Win32 ported code) using the same test system from UNIX by running more than 1000 test flows in a seamless mixed NT-UNIX platform. The test flows require execution of both the IA-NT ported CAD tools as well as the UNIX dependency CAD tools that are not yet available on Windows NT (MTEX enables runtime client-server modeling).

We have also received encouraging feedback from our UNIX CAD developers who participated in the early IA-NT productization and deployment effort. With MTEX, they are able to pick up the IA-NT ported codes for their tools from migration engineers and integrate back into their environment with little effort. They can use the pre-existing test suite and test system to validate the correctness of the ported codes. With no environment change, the deployment task to IA-NT has proven to have negligible impact to their daily work.

These CAD engineers appreciate the flexibility of the MTEX that enabled them to combine tools from both NT and UNIX environment to form a single seamless platform independent design flow execution environment. Having a familiar tool working under Windows NT, it would be a good starting ground for them to pick up Windows NT knowledge and to work on future improvements to their tool environment to take advantage of native Windows NT features.

8. Limitations & Future Work

The limitations of current MTEX version are:

- Scripts need to be modified to remove all references to local paths such as /tmp since the mixed UNIX and NT environment depends on the shared file system for their data exchange.
- If application processes communicate with each other through IPC, all of these processes need to be executed on the same platform.
- For MTEX to work correctly, the user has to be logged on to the NT system.

The next version of MTEX will offer the following features:

- Utilities to display MTEX invoked processes and to terminate MTEX invoked processes from any system where MTEX is installed.
- Removal of the limitation of the user to be logged on to the NT system for MTEX to work correctly.
- Conversion rules moves to the server end so that support for a new platform can be added on the fly.
- GUI interface for MTEX management.

9. Conclusion

The Multi-platform Tool Execution eXtensions (MTEX) technology is a simple but powerful technique to provide a transition path, a bridge to migrate a very complicated UNIX-centric environment to Windows NT.

MTEX is by no means THE solution to the UNIX scripting problems on Windows NT. However, we believe that MTEX is a very important transition technology to enable a successful UNIX to NT migration. By supporting a platform independent runtime environment, MTEX is implicitly enforcing the classic computer science "divide and conquer" strategy. What seems to be an impossible mission of moving all Intel CAD design tools, CAD runtime and test environment, user training, etc. from UNIX to Windows NT turns out to be a feasible project by using MTEX. The reader can refer to our paper in Intel Technology Journal (Q1/99 issue, details in reference section) for another innovative compile-time client-server technology for their mixed NT-UNIX applications.

Our motivation for writing this paper is to share our experience on an innovative transition technology we engineered to meet our CAD migration needs. MTEX is generic and can be easily proliferable to any other environments in any organization. However, we believe that the best migration strategy is the one that have been carefully studied and evaluated according to the specific requirement of the organization.

10. Acknowledgements

We thank Intel management for providing the opportunity to work on this exciting project. We are grateful to our department manager Tae Paik for his vision, inspiration, patience, and continuous encouragement during this project. We wish to acknowledge Tae Paik for the brainstorming session that helped us to come up with a run-time client-server model. Thanks to Athena-NT Technical Working Group members for ratifying the technical concepts and documents. Special thanks to Tzvi Melamed, Greg Hannon, Chenwei Chiu, and Ming Lin for their technical feedback and encouragement during the development work. We would like to acknowledge Rumi Zahir for motivating us to write this paper.

11. Trademarks

All brand names are the property of their respective owners.

12. References

[1] Shesha Krishnapura et al. "CAD Design Flows Development in a Cross-Platform Computing Environment", Intel Technology Journal Q1 1999.

[2] Alexander Wolfe, "Intel taps Windows NT in design-software shift." EETIMES, issue 948, April 7, 1997, pages 1, 148.

[3] Richard Goering, "Can NT win in IC design?" EETIMES, issue 992, page 70.

[4] Jeffrey Richter, Advanced Windows (3rd Ed) Microsoft Press 1997.

[5] Marshall Brain, Win 32 System Services: The Heart of Windows 95 and Windows NT. Prentice Hall 1995.

[6] Andrew Lowe, Porting Unix Applications to Windows NT. Macmillan 1997

[7] David A. Solomon, Inside Windows NT (2nd Ed). Microsoft Press 1998.

[8] Jeffrey Richter, Win32 Q&A. Microsoft Systems Journal June 1998.

[9] Jeffrey Richter, Manipulate Windows NT Services by Writing a Service Control Program, Microsoft Systems Journal February 1998.

[10] Jeffrey Richter, Design a Windows NT Service to Exploit Special Operating System Facilities, Microsoft Systems Journal October 1997.

Porting Legacy Engineering Applications onto Distributed NT Systems.

N.K. Allsopp, T.P. Cooper, P. Ftakas
Parallel Applications Centre, 2 Venture Road, Chilworth, Southampton SO16 7NP
P.C. Macey
SER Systems Ltd, 39 Nottingham Rd., Stapleford, Nottingham NG9 8AD

Abstract

In this paper we present our experiences developing two distributed computing applications on NT. In both examples a legacy application is ported from Unix to NT and is then further developed to be a distributed application within the NT environment. We shall present two different approaches to managing the remote execution of tasks. One is a port of a serial vibroacoustic analysis code called PAFEC VibroAcoustic and the other is the parallelisation of the non-linear analysis modules of the LUSAS FE analysis package. We shall show in these two projects that it is technically possible to carry out scientific computing on a distributed NT resource

1. Introduction

In this paper we present our experiences developing two distributed computing applications on NT. In both examples a legacy application is ported from Unix to NT and is then further developed to be a distributed application within the NT environment. We shall present two different approaches to managing the remote execution of tasks. For both applications we shall present performance metrics and discuss the benefits of running distributed applications on NT.

The first application is a port of a serial acoustic analysis code, PAFEC VibroAcoustic, from Unix to run in parallel on a cluster of NT workstations. The port was funded by the European Union in the project PACAN-D. The University of Southampton Parallel Applications Centre (PAC) and SER Systems Ltd (code owners) carried out the parallelisation of the code before being assessed by an industrial end-user, Celestion International. Celestion are a small manufacturing company who design and build loudspeakers for home entertainment. As a small concern wishing to minimise costs, they were obviously attracted to NT. They became involved in the project to determine the conditions under which their cluster of desktop machines could also be used for running numerical simulations. In particular they wanted to test whether the cluster could be fully dual use, or whether they would still need to invest in extra computing resource to serve their simulations. The parallelisation of the code was implemented using MPI, enabling the same source to be used for NT as for Unix applications. Celestion tested the code on their cluster of single processor NT machines. The machines were dual use, in that they were used for other tasks during the day and were available for execution of large tasks over night.

The second application is the parallelisation of the non-linear analysis modules of the LUSAS FE analysis package from FEA Ltd. The code was well suited to a domain decomposition approach, and a major part of the effort in the project was the porting to NT of an intelligent resource manager (Intrepid), initially developed by PAC for heterogeneous clusters of Unix workstations. The issues that had to be addressed in performing this task were wide ranging. The Intrepid code was over 70000 lines of C and C++ and utilised a number of Unix tools to compile, not all of which were available on NT. In addition the functionality on which a resource manager relies, such as the methods of controlling remote execution, of monitoring tasks and of copying data sets all had to be completely redesigned. The work was funded by the EU as part of the project PARACOMP and evaluated by Messier-Dowty on a cluster of NT workstations. Again these machines were dual use (although unlike Celestion there was some spare capacity). One other feature of this cluster was its heterogeneity; machines varied between 166Mhz and 400Mhz clock speed, with a similar variation in memory and disk performance. One issue that had to be addressed is that of controlling access to machines. A problem that takes just over 4 days to solve in serial may only take 1 day to run on

4 processors. However if it prevents engineers from working on those machines for that time, there is no increase in productivity. The issue for Messier-Dowty was to have a code that would deliver a result in the same elapsed time, but would utilise the parallel speed-up to enable it to run only overnight.

2. Background to Projects

2.1 The PACAN-D Project

The aim of the PACAN-D project was to deploy a parallel version of the PAFEC VibroAcoustic finite element analysis package in a loudspeaker business. The work was based upon an existing parallel PAFEC VibroAcoustic code that was developed in a previous collaboration between SER Systems Ltd and the PAC. This code was developed in 1993 before the appearance of effective standard message passing interfaces. The code was also specific to the Intel iPSC/860 and Paragon platforms.

The objectives of the port can therefore be summarised as to port the old parallel PAFEC VibroAcoustic code from the Intel iPSC/860 to modern parallel systems and standards. As the code is under constant development it was decided to consolidate the parallel code with the latest version. As the code was originally developed on a UNIX based operating system and to reduce the number of platform dependant version of the code it was decided to select a message passing protocol, which could be easily used on different platforms. Therefore it was decided to use MPI for the message passing.

The PAFEC VibroAcoustic system is mainly written in FORTRAN, but some of the low level machine dependent parts are written in C. There are several hundred thousand lines of FORTRAN code. Original sections of the code were written in FORTRAN IV. More recently FORTRAN 77 and FORTRAN 90 have been used. The system is still being actively developed. There are many different executables in the system, some for running different stages of the analysis, and some for converting dictionary files from ASCII to binary form. It is possible to run jobs including user supplied FORTRAN routines, either as an efficient way of specifying data or to use a modified version of a standard system routine. Under these conditions the system runs a shellscript to perform a compile and link for the appropriate executable, while running the analysis. The system is currently developed in a Unix environment on HP workstations.

2.2 The PARACOMP Project

The PARACOMP project was an ESPRIT-TTN project whose main aim was "to demonstrate the deployment of a parallel code to perform composites analysis on a network of NT workstations, and to disseminate the benefits". Three companies were involved in the project:

- FEA Ltd. supplied the finite element analysis code for the project. They implemented a parallel solver, which was based on their legacy FORTRAN finite element analysis solver called Lusas.

- Parallel Applications Centre supplied the resource management system and the integration software for the project. An intelligent resource manager called Intrepid and developed by the PAC was used in the project. Intrepid was originally written for the UNIX operating system, but it has been ported to Windows NT for this project.

- Messier-Dowty tested the software for the analysis of composite materials.

The Intrepid parallel scheduler is a tool that allows the scheduling, control and execution of a number of tasks (programs) on a heterogeneous network of workstations. It allows the user to control both the resources used by the scheduler to run programs on, as well as the programs running themselves.

The general structure of the Intrepid parallel scheduler ("set in stone" by the Unix version of the software) is shown in figure 1.

As can be seen from the figure, the Intrepid parallel scheduler consists of two main components:

pacS: This is the main scheduler executable and the program that contains the global network and schedule information for the system. This module takes all the scheduling decisions in Intrepid. There is only one instance of pacS running in a functioning Intrepid system.

pacD: This is the daemon that provides support to the Intrepid parallel scheduler. There is one of these daemons in each "node" in the network. An Intrepid daemon's main function is to pass back information to pacS about the state of the "node" which it is controlling, and to launch and monitor applications. Secondary functionality includes all the necessary actions to start an application (e.g. creation of temporary di-

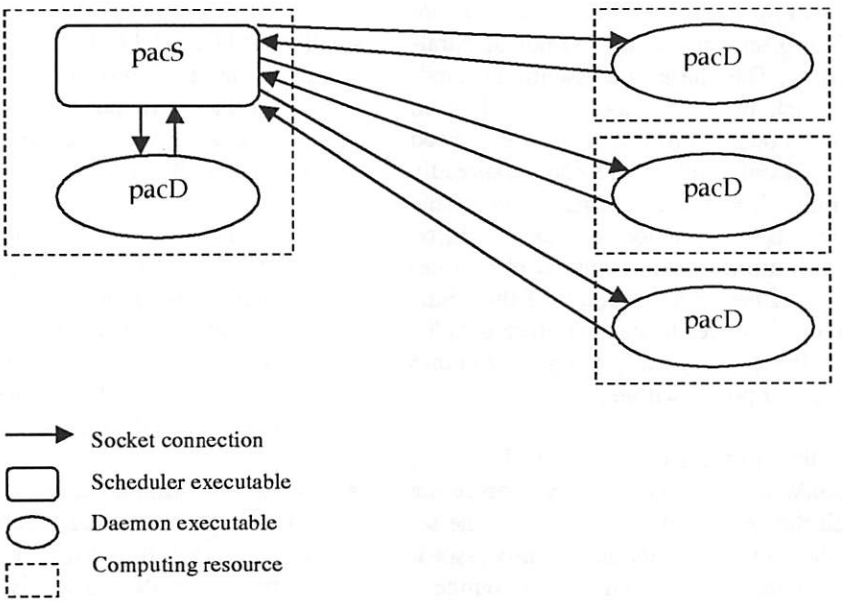


Figure 1: General Structure of the Intrepid system.

rectories, if required, transfer of all input files before and output files after the execution, etc.).

Communications between pacS and the various pacD daemons are implemented via TCP/IP sockets. This allows Intrepid to control heterogeneous networks. The user of Intrepid submits a "schedule" to the system. A schedule consists of a number of tasks (processes) to be executed and the dependencies between them. The Intrepid system then assigns tasks to specific processors in the network, so that the overall time of execution for the whole schedule is minimal. The system also performs any data transfers that might be required to allow a task to run on a specific machine. Any number of users can submit any number of schedules to Intrepid simultaneously and the system is able to deal with the added complexity.

3. Techniques Used

3.1 The PACAN-D Port

When porting onto the PC with the Digital Visual FORTRAN compiler it was found to be very time consuming to set up the visual environment. Indeed it was not clear how best this should be done for a system containing thousands of routines and with multiple executables. Attempts at setting up the environment were hampered by a bug in IE3, which has since been fixed, which prevented large numbers of files being copied at once. Consequently everything was done using command shells. This did not help at a later stage with debugging the test jobs which failed, as less information was available on error conditions. There are limits on the command line length, but the shells generated in Unix ports depended on long lines to link in all the appropriate libraries. Thus it was necessary to use resource files rather than wild cards and this necessitated a fundamental change to the structure of one of the dictionary files. Problems were encountered with handling sequential files, which seem to be compiler dependent. Furthermore there were problems with opening files, which did not work in dynamic link libraries. The machine dependent FORTRAN parts were handled by a rewrite of routines from a previous PC port, which used a DOS extender, and worked using DOS interrupts. The Digital Visual FORTRAN compiler was relatively strict, compared with those used in many Unix ports, and detected some occurrences of inconsistent numbers of parameters in routine calls. As always code errors found were fixed in the development level, making the code more robust and portable for the future.

The PAFEC VibroAcoustic system consists of a suite of programs called *phases*. There are 10 phases in total, phase 1 to 6 performs the pre-processing of the element data of the model to be analysed. Phase 7 carried out the solution of the equations, the numerically intensive part of the whole system. Phases 8 to 10 handle the plotting and visualisation of the calculated

data. Information passes between the phases via files held on disk. It was apparent that this phase structure of the PAFEC VibroAcoustic code need not be parallelised in its entirety. The phases are essentially standalone programs each of which may be parallelised independently of the others. Profiling the code showed that phase 7 carried out most of the computationally intensive operations. The most striking result of the profiling exercise was that a majority of the execution time was spent within a very small number of routines i.e. 50 routines out of the 18000 routines of the whole PAFEC VibroAcoustic code. It was therefore decided to concentrate on these numerically intensive routines which occurred at four points within phase 7.

The underlying ethos to the parallelism of phase 7 of the PAFEC VibroAcoustic is that the master processor proceeds through the code in the same way as the serial version. At the point where the master processor is about to enter a numerically intensive subroutine a message is sent to the other slave processors to indicate which routine is being entered. On entering the numerically intensive routine, all the processors perform an equal amount of the required calculation. When the routine is finished the master processor continues to progress on through the serial code whilst the slave processors wait for the next numerically intensive section to be reached by the master processor.

For a fully coupled vibroacoustic solution, using an acoustic BE mesh coupled to a mesh of structural FE the set of equations to be solved can be written as:

$$\begin{bmatrix} [S] & [T] \\ [G] \quad [E] & [H] \end{bmatrix} \begin{Bmatrix} \{U\} \\ \{p\} \end{Bmatrix} = \begin{Bmatrix} \{F\} \\ \{p_I\} \end{Bmatrix}$$

Where:

[S] Contains the structural stiffness matrices which are large and sparse,

[H], [G] are small dense matrices derived from the BE formulation.

[T], [E] are coupling matrices.

Where {u} is a vector of displacements on the structural mesh, {p} is a vector of pressures on the BE. [S], [C] and [M] are the structural stiffness, damping and mass matrices and are large and sparse. [H] and [G] are small dense matrices derived from the BE formulation. [T] and [E] are coupling matrices. Sometimes the structural representation is simplified using a smaller modal model of the structure, but this does not permit variation of properties with frequency, which occurs for the surround and cone on a loudspeaker. The current work was based on a full solution of the above equation, using the four stages below.

- Stage 1 - FE merging/reduction. The dynamic stiffness matrix and coupling matrix [T] are formed by merging contributions from individual finite elements. The equations are simultaneously solved. Degrees of freedom are eliminated as early as possible. The matrices are shared between processors and the elimination is done in parallel.

- Stage 2 - forming the BE matrices. For each collocation point on the BE surface it is necessary to integrate over the surface to form a row in the BE matrices. Parallelization is achieved by sharing these collocation points between the processors. Distributed BE matrices are formed.

- Stage 3 - reducing the BE matrices. The matrix $-\omega^2\rho[G][E]'$ is formed and reduced using resolution with the structural elimination equations from stage 1. As above the matrices are distributed between processors.

- Stage 4 - Gaussian elimination of final equations. The resulting compact dense set of equations is solved using a parallelized form of Gaussian elimination on the distributed matrix.

3.2 The PARACOMP Port

3.2.1 Porting Intrepid to Windows NT

The Intrepid system was originally implemented on UNIX systems. The dependencies of the source code on libraries and development tools were kept minimal (even in the UNIX world there are a large number of different flavours of UNIX and Intrepid was designed to be portable). Intrepid relied on the following external dependencies:

- C++ compiler. In UNIX, native C++ compilers on the corresponding platforms were used to compile Intrepid. Although the code is not ANSI C, or POSIX compliant, the transfer from UNIX to Visual C++ did not present large difficulties, as the main bulk of the code is written as a console application and does not rely on any GUI functions.

The UNIX GUI to Intrepid (which existed in Tcl/Tk and X/Motif incarnations) has been lost with the transit to Windows NT.

- Socket library. The windows socket library presented minimal problems during the porting process. Problems were mostly solved with the use of C pre-processor macros to distinguish between dissimilar UNIX and Windows32 API socket function calls and constants. The reason that it was decided to use sockets for communications between the various components of Intrepid was historical. Sockets are the de-facto standard for fast low-level communications in all UNIX systems. The use of sockets though, had another consequence: the Intrepid resource manager can be used to control a heterogeneous network of workstations, running different operating systems and not having access to a common, uniform file system. The system is able to distinguish between a number of different operating system types and treat workstations running those operating systems accordingly. It also implements file transfers for input and output files (through sockets, or using the Windows32 drive mapping mechanism and file copying functions to perform the transfer).

- Environment variables. Although environment variables exist in the case of Windows NT as well as UNIX, a cleaner solution in Windows NT would be to have application information held in the registry and not in environment variables. This was partially implemented during the PARACOMP project. Although all the environment variables are available through the registry, they have been introduced under the HKEY_LOCAL_MACHINE key. Since Intrepid is a multi-user system, this has security implications for the system, since all users will see the same values for the environment variables. In UNIX, Intrepid relies on various shell scripts being executed at login for the user, in order to correctly implement security.

- rshd. At start-up, the Intrepid system has to start (daemon) processes on every workstation that will be controlled by the system. In the case of UNIX machines, as long as the workstation is reachable and the username is accepted, Intrepid can execute a remote shell command on the remote machine and start up the (daemon) processes. Unfortunately, Windows NT does not come with a remote shell daemon (rshd) as standard, and most system administrators are not willing to give such powers to their users. The solution we have come up with was to implement these (daemon) processes as Windows NT services. Instead of Intrepid explicitly spawning these processes at start-up, these processes always exist on the background, waiting to connect to the resource management system. In reality they are blocked listening on a socket, so impact on system resources is minimal while Intrepid is not running.

- lex/yacc. Part of the Intrepid system was a parser that was written using lex and yacc. The only versions of lex and yacc that we could have access to on Windows NT would be the GNU implementations. However, the UNIX code was written with the BSD versions of lex and yacc, which has some slight differences. The solution we used was to execute the BSD lex and yacc tool implementations on a UNIX workstation to create the C code for the parser. We then performed some minor changes to the included files and definitions in the source code (a program was written to do this automatically) and finally included the resulting code in the VC++ project for Intrepid.

- Data transfers. The Intrepid version of UNIX used two different methods to perform data transfers: direct copying using the standard UNIX copy command cp, or a pair of helper programs that are used to implement the data transfer using sockets. The former method cannot be used in NT since it presumes the existence of NFS for it to work. The latter method is generic enough to work in the context of a Windows NT environment. However, after the porting of helper programs from UNIX to NT, we found that the system was too slow. We implemented an alternative data transferring mechanism that uses drive mappings to effect the file copy. In Windows NT, one computer can have access to the file store of another computer through the drive mapping mechanism. There is a notion of network file names called by Microsoft UNC (Universal Naming Convention). However, these UNC file names cannot be used directly with the Win32 API functions that handle files[5]. The solution Intrepid has come up with, is to map the UNC directory to a free drive letter and then use a normal file path (containing the new drive letter, of course) to perform any operations on the file. Once the file is copied to a local execution directory the drive letter can be unmapped, so that it can be used again on some other file transfer. The limitation of this method is that the system administrator must set up the system in such a way

that the products that are defined in the task graph file point to the intended files. This means that if a product is defined to consist of a file at location "\\machine\dir1\file.dat", then the directory **dir1** must be shared on the workstation named **machine.**

3.2.2 Interfacing Intrepid with LUSAS

Lusas consists of a number of components. The two main ones are the solver itself (Lusas solver) which is the legacy Fortran code for the finite element analysis, and the GUI front end (Lusas modeller) which is a Windows32 application and is used to design the model and present results.

The usual mode of operation for Lusas is that a user will design a model using the modeller, use the solver on the created model and finally display the results on the modeller window. To start the solver on a model all a user has to do is click a button.

We wanted to keep the same style of operations in PARACOMP. However, there are a number of steps involved in generating the parallel solution that have to be hidden from the user. When a model is ready to submit to the parallel solver, the first thing that PARACOMP must do is split the model into the appropriate number of sub-domains in order to perform the finite element analysis in parallel. Appropriate data files have to be generated for each of the sub-domains of the model. An input file to Intrepid has to be generated (the task graph file) that contains information about the dependencies between the composite parts of the finite element analysis. For each of the sub-domains a different process running the parallel solver will have to be spawned on the target machine. Before the process is started the necessary data files that represent the domain have to be copied over to a local directory on the target machine. When the finite element analysis run is finished the result files have to be copied back to the machine from which the user submitted the job. All this is handled by the Intrepid system and a small Visual Basic script that performs the domain decomposition and essentially "glues" Lusas and Intrepid together [5]. All the user input required is the number of sub-domains that the model must be split into.

There is also a need for another GUI program, which would configure Intrepid for the specific network that Intrepid is used on. Because of time limitations and because PARACOMP only addressed networks of Windows NT workstations, the software that performs this function does not have the full functionality of the UNIX GUI for setting up Intrepid. It assumes that the network only contains NT machines and the options that the user can set at configuration time are comparatively limited.

4. Results

4.1 PACAN-D

The following results were obtained using the test case supplied by Celestion International. The size of the test case represented the maximum size of problem that could be simulated on their existing single machine. This particular test case is representative of a typical loudspeaker system under test at Celestion. The system represented a half model, its parameters are 141 structural elements, 1917 structural degree of freedom and 914 acoustics degrees of freedom (interior and exterior).

This test case was run on a cluster of 166 MHz Pentium Pro machines linked together with standard 10Mbit Ethernet. To reduce traffic conflicts over the Ethernet the cluster were linked via a switch that effectively isolated the cluster from the rest of the network. The same test case was then run using the same code but compiled with a different version of MPI[1,4] on a shared memory SGI machine consisting of eight 75MHz processors although only 4 processors were used.

Stage	1 Proc.	2 Proc.s	3 Proc.s	4 Proc.s
1	73	74	73	75
2	165	86	58	41
3	975	484	315	227
4	132	95	85	104
Total	1345	739	531	447

Table 1: Results on 75MHz SGI Machine using MPICH.

Stage	1 Proc.	2 Proc.s	3 Proc.s	4 Proc.s
1	49	56	66	70
2	81	42	31	18
3	508	286	173	134
4	76	84	100	96
Total	714	468	369	318

Table 2: Results on 166MHz Pentium Pro using WMPI 1.01.

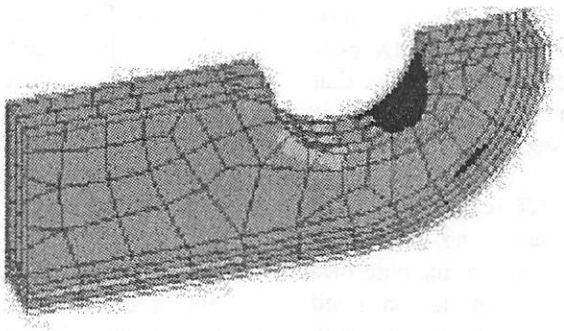


Figure 2: Landing gear component from Messier-Dowty.

4.2 PARACOMP

The final stage of the PARACOMP project involved the use and evaluation of the final software by an end user, which was Messier-Dowty. The software was installed and run on a model defined by Messier-Dowty. The model is one half of a lug which is the main load bearing component of a landing gear. The load is transmitted in plane through the landing gear and up through the lug into the wing. The aim of the investigation was to identify the onset of delamination in the lug, and thence provide input into the type of physical testing to be carried out.

The experiment consisted of the parallel finite element analysis of a design decomposed into two sub-domains. The relative size of the two sub-domains is important since the target machines had different processing power. One (which we will name machine A) was a 400MHz Pentium II machine and the other (machine B) was a 166MHz Pentium. The time taken to execute the full sequential solution on machine A was 46 minutes and on machine B 110 minutes. If we assume that the execution time for a sub-domain on a processor is a linear function of the size of the sub-domain, then we can calculate that the optimal run for the parallel case is 33 minutes. This can be seen as the minimum value of the execution time curve in figure 3. When the software was used to perform a parallel run on the same model, the actual execution time was 29 minutes. We think that such a discrepancy exists because the code is I/O bound. By distributing the I/O between the two machines, we achieve a super-linear speed-up.

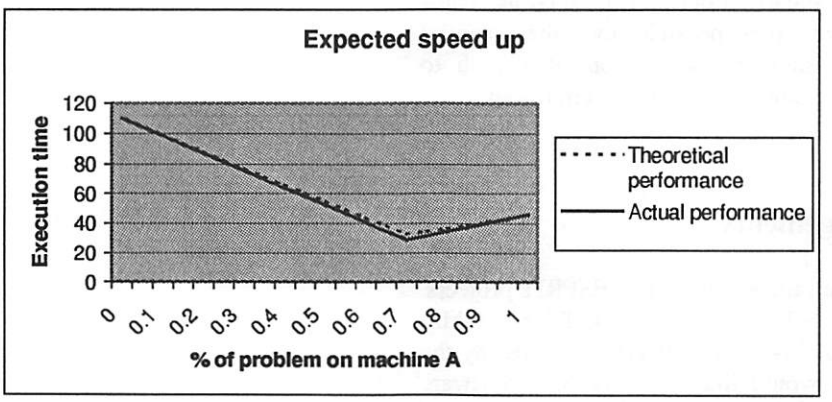


Figure 3: Results on a heterogeneous cluster of a 400MHz Pentium II and a 166MHz Pentium Pro.

5. Conclusion

The MPI protocol has been previously proven to provide portability of source code between UNIX platforms. In the PACAN-D project we have shown that this portability extends to NT, and that performance is sufficient to yield satisfactory speed-ups.

One of the main concerns about NT is that absence of remote shell procedures make controlling the remote execution of jobs difficult. However the mapping of a Unix daemon to an NT service has been demonstrated in the port of Intrepid to NT. In addition it is worth noting that the MPI daemons are also implemented as NT services, and it is our conclusion that the NT service functionality provides a robust way of managing remote execution in the NT environment.

Three methods of data transfer between machines have been tested: using WMPI calls, using sockets and using drive mappings to move files. The WMPI layer is fast enough to support serious numerical computation. A comparison between the use of sockets and drive mappings was made as part of the Intrepid port, and drive mappings were found to be faster. It is not clear whether this is fundamental to the design of sockets on NT, or because of the implementation of the transfer within Intrepid.

With the ever-increasing processor speed of NT platform coupled with the decreasing unit cost, the proposition of porting legacy commercial codes to NT is increasingly attractive. We have shown in these two projects that it is technically possible to carry out scientific computing on a distributed NT resource. The next stage of the work will be to assess whether this computation can be carried out on truly dual use hardware. If this proves to be possible, then the potential cost benefits for smaller organisations that wish to invest in numerical simulation become enormous.

6. Acknowledgements

This work has been funded as part of ESPRIT projects, no. 24871 PACAN-D and no. 24474, PARACOMP, both part of the HPCN TTN Network supported by the EC. The authors would like to thank MPI Software Technology Inc. and Genias Gmbh for their support of this research

7. References

[1] Message Passing Interface Forum, MPI: A Message-Passing Interface Standard, May 5, 1994, University of Tennessee, Knoxville, Report No. CS-94-230

[2] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, A high-performance, portable implementation of the MPI message passing interface standard.

[3] W. Gropp and B. Smith, Chameleon parallel programming tools users manual. Technical Report ANL-93/23, Argonne National Laboratory, March 1993.

[4] J.M.M. Marinho, Instituto supererior de Engenharia de Coimbra, Portugal.

[5] Sing Li, Professional Visual C++ ActiveX/COM Control Programming, Wrox Press Ltd, 1997.

[6] Microsoft Corporation, Moving UNIX Applications to Windows NT, version 0.9 (part of Visual C++ online documentation), 1994.

[7] Meecham, K; Floros, N; Surridge, M. Industrial Stochastic Simulations on a European Meta-Computer, Proceedings 4th International EuroPar Conference, EuroPar 98 Parallel Processing.

[8] Cooper, T. Case studies of 4 industrial meta-applications, Proceedings HPCN Europe 99.

Porting a User-Level Communication Architecture to NT: Experiences and Performance

Yuqun Chen, Stefanos N. Damianakis, Sanjeev Kumar, Xiang Yu, and Kai Li
Department of Computer Science
Princeton University, Princeton, NJ-08544
{yuqun, snd, skumar, xyu, li}@cs.princeton.edu

Abstract

This paper describes our experiences in porting the VMMC user-level communication architecture from Linux to Windows NT. The original Linux implementation required no operating system changes and was done entirely using device drivers and user-level libraries. Porting the Linux implementation to NT was fairly straightforward and required no kernel modifications. Our measurements show that the performance of both platforms is fairly similar for the common data transfer operations because they bypass the OS. But Linux performs better than NT on operations that require OS support.

1 Introduction

The primary goal of the SHRIMP project is to investigate how to design high-performance servers by leveraging commodity PC hardware and commodity PC software. A key research component is to design and implement a communication mechanism whose performance is competitive with or better than that of custom-designed multicomputers. The challenge is to achieve good communication performance without requiring special operating system support. This paper describes our design decisions, experience, and performance results of porting our virtual memory-mapped communication (VMMC) mechanism and stream sockets to Windows NT.

Our previous work on fast communication for PC clusters was done for the Linux operating system. During the initial phase of the project, we studied how to design a network interface to support VMMC and successfully implemented a 16-node prototype PC cluster with our custom-designed network interfaces and an Intel Paragon routing network [5, 7, 6]. The VMMC mechanism, our low-level communication layer, performs direct data transfers between virtual memory address spaces, bypassing the operating system. Its main ideas are separating control from data transfer and using virtual memory mechanism to provide full protection in a multi-programming environment. We demonstrated that VMMC on Myrinet achieves 3.75 μs end-to-end latency using a custom-designed network interface.

During the second phase, we designed and implemented a VMMC mechanism with extended features for PC clusters connected via Myrinet, a commercial system area network and the operating system was again Linux [15, 9]. With a programmable network interface (Myrinet), we showed that VMMC achieves about 14 μs end-to-end latency and delivers bandwidth close to the hardware limit. We also designed and implemented several compatibility communication layers such as message-passing [1], RPC [3], and Unix stream sockets [14], and showed that they deliver good performance.

We recently ported VMMC and several compatibility software libraries to PCs running Windows NT 4.0 and 5.0 Beta. Several factors motivated our entry into the NT world. First, we want to leverage more PC hardware devices and software systems that are available for the NT platform. We particularly want to use high performance graphics cards that have only NT drivers. Second, we needed good kernel support for SMPs. Linux did not have such solid support for multiprocessors. The ever changing nature of Linux kernel sources also poses a daunting task for maintaining our VMMC software up-to-date. It also makes it really difficult to distribute our software to other research institutes. Porting VMMC to Windows NT eliminates this concern and allows us to take advantage of the huge NT user base.

This paper describes our design decisions, porting experience, and performance results. We also address the questions, whether the promise of requiring no special OS support still holds when port-

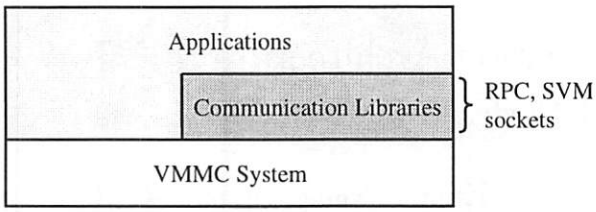


Figure 1: Communication Architecture

ing the communication mechanisms to NT, and whether the porting retains good communication performance. Lastly we would like to share our experience with the readers about the pros and cons of porting user-level communication mechanisms to Windows NT.

2 Communication Architecture

The communication architecture developed in the SHRIMP project (Figure 1) consists of two layers: (i) high-level communication libraries, and (ii) virtual memory-mapped communication (VMMC) mechanisms. Applications such as scientific computation, parallel rendering, and storage servers can use the VMMC primitives directly or use the high-level communication libraries.

The VMMC layer consists of a set of simple, yet powerful and efficient communication primitives for protected, user-level communication in a multi-programming environment. The basic idea of VMMC is to transfer data directly between virtual address spaces across a network with minimal overhead. The approach taken was to provide a mechanism to set up protected communication among virtual memory address spaces across a network, and a separate mechanism to initiate direct data transfers between virtual address spaces at user level efficiently, conveniently, and reliably. The VMMC layer is system dependent. It provides very low-overhead communication with network interface hardware support [5, 7] and quite efficient communication with a programmable network interface [15].

The high-level communication libraries take advantage of VMMC primitives to support applications which use legacy or complex communication APIs. In the SHRIMP project, we implemented Unix stream sockets [14], remote procedure call [4], NX message-passing library [1], and various shared virtual memory systems [17, 21, 6]. Critical to our layered approach is the requirement that the underlying VMMC layer provides convenient mechanisms for the high-level communication libraries to implement zero-copy protocols, in particular, connection-oriented communication protocols. It is also important to consider the tradeoffs as to which layer implements reliable communication. Our approach was to implement a retransmission protocol at VMMC layer, so that we can achieve low-latency and high-bandwidth reliable communication and simplify the construction of high-level libraries such as stream sockets.

The main rationale of the two-layer communication architecture is to minimize system dependence. Because the communication libraries sit directly on top of VMMC, they are system independent. Our porting of the communication architecture from Linux-based PC clusters to Windows NT clusters validated our design rationale.

3 Porting VMMC to NT

In this section, we first describe the components of VMMC, then discuss porting issues, and finally report the lessons that we learned from our porting experience.

3.1 VMMC Components

The VMMC layer consists of three sets of primitives. The first set contains the primitives for setting up VMMC communication between virtual address spaces in a PC cluster. The primitives include `import` and `export` as well as `unimport` and `unexport` of communication buffers in virtual address spaces. Implementation of these primitives requires system calls because they need access to information about memory pages and permission checkings. These primitives are intended for applications to use during communication setup phase. In general their performance is not very critical.

The second set of primitives are for data transfer between virtual address spaces. They include synchronous and asynchronous ways to send data from a local virtual memory buffer (VM buffer) to an imported remote VM buffer, and to fetch data from an imported remote VM buffer into a local VM buffer. Initiation of these primitives is done entirely at user level, after the protection has been set up via the setup primitives. Thus data transfers are very fast and also fully protected in a multi-programming environment. An optional notification mechanism allows a data transfer primitive to invoke a user-level handler in the remote process, upon transfer completion. VMMC also includes primitives for redirecting incoming data into a new VM buffer in order to maximize the opportunity to avoid data copy when implementing high-level communication libraries.

The third set of primitives are utility calls for applications to get information about the communication layer and the low-level system. The primitives also include calls to create remote processes, and obtain and translate node and process ids.

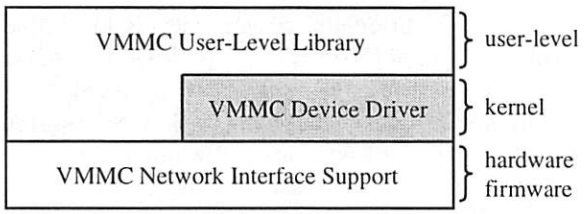


Figure 2: The VMMC System Architecture

Figure 2 shows our initial implementation of VMMC for a Myrinet-based PC cluster which runs on Linux OS. This implementation consists of three components: the network interface (NI) firmware, a device driver, and a user-level library. The NI firmware (also called Myrinet Control Program or MCP) implements a small set of hardware commands for user-level, protected VMMC communication. The device driver initializes the NI hardware, downloads the MCP, and performs firmware initializations. The device driver also implements the setup primitives of VMMC. The user-level library implements all data transfer primitives. For each virtual address space, there is a memory-mapped command buffer for the user program to initiate data transfer commands at user level.

A unique feature of the VMMC is the use of a User-managed TLB (UTLB) to perform address translation. The UTLB mechanism does demand page-pinning. It pins a local buffer when it is used in communication for the first time. Subsequent data transfers using the same buffer will be able to translate addresses efficiently and safely at user level. The UTLB mechanism may unpin the buffer according to some memory allocation strategy. For applications that display spatial locality in their communication patterns, the cost to pin and unpin the virtual pages is amortized over multiple communication requests. A recent paper [9] provides the details about the design, implementation, and evaluation of UTLB.

In addition, the VMMC also implements a retransmission protocol at the data link level and a dynamic network topology mapping mechanism to provide high-level libraries and programs with a reliable low-level communication layer.

3.2 Porting to NT

Although NT is quite different from Linux, the architecture components of the Linux VMMC implementation fit NT quite well. We therefore maintained the original structure. However, we did need to make several changes besides restructuring the device driver for NT.

The first is to use NT kernel threads to enable VMMC for shared memory multiprocessor (SMP) nodes. The original VMMC for Linux did not use threads, because until recently Linux did not officially support threads. Because VMMC provides protected communication in a multi-programming environment, our SMP clusters using Linux used multiple address spaces on each node to implement communication libraries such as SVM [21]. The main drawback of this approach is that the cost of a process context switch is much higher than a kernel thread context switch. During the NT port, we used NT events, semaphores and conditional variables to perform synchronizations for VMMC calls so that multi-threaded user programs can safely use VMMC without explicitly performing synchronizations.

The second is to use NT kernel threads to implement notifications. The Linux version uses Unix signal to implement a notification. After the NI DMAs a packet into the host memory, it puts the the virtual address and the value of the last word in the data into a message queue, which is shared by driver and mcp, and then triggers an interrupt to the host processor. On Linux, the host interrupt handler directly sends a signal to the receiving process. The kernel schedules the signal handler to run in the user process' context. NT does not provide Unix-style signals. Instead, the host interrupt handler (actually a Deferred Procedure Call or DPC handler) triggers an NT event on which a dedicated notification thread is waiting. Upon wakeup, the notification thread calls the appropriate handler in user context. Our NT port also allows a thread to wait on an event explicitly, bypassing the notification handler mechanism. The event can allocated for each exported buffer. This is a useful feature for multi-threaded applications, because each thread can explicitly wait on its own messages.

The third is to deal with remote process creation. The Linux version uses `rsh` to spawn a remote process. However, we cannot use this method because, at the time of porting, we were not aware of any method to specify a working directory for a remotely launched program (on a central file server). To get around this restriction, remote pro-

cess creation is handled by a service running on each node in the cluster. The master process uses RPC to talk to the remote service to create a process. The service then maps the remote directory to a local drive letter and starts the program. We also implemented cluster management such as redirecting the output of each process to central file system, through the service,

3.3 Lessons Learned

Windows NT provides more mature support for device driver development than Linux. NT has complete documentation, many example sources, and a good kernel debugger to work with. But, since we do not have NT kernel source code, sometimes it is difficult to pinpoint bugs in the driver. On Linux, there is little documentation on device drivers; we had to study existing device drivers and kernel source code. Further, we had to modify the kernel source to export more symbols that were needed for our VMMC device driver.

NT supports SMP and provides a set of kernel primitives to support thread synchronization. However, they can not be used in all places, ie. interrupt handler, kernel DPC handler, and device IOCTL can use different subset of primitives. It took us some time to figure out which primitives to use.

Much of the porting effort was focused on making VMMC thread-safe to take advantage of NT's kernel threads. For example, our Linux SVM system used multiple processes on each SMP, while our newer NT SVM uses a single process with multiple kernel threads. Using kernel threads resulted in a significant performance improvement for SVM.

4 Porting Sockets to NT

The user-level stream sockets library allows applications based on stream sockets to leverage our fast communication substrate with no code modification. This library was initially developed for VMMC Linux clusters. In this section we first describe the sockets model and how it differs on Windows NT. Then we discuss the issues involved in implementing Winsock library using the Unix stream sockets as the base. We end this section with the lessons learned during the porting process.

4.1 Stream Sockets

A wide variety of distributed applications rely on the Berkeley Unix stream sockets model for interprocess communication [18]. The stream socket interface provides a connection-oriented, bidirectional byte-stream abstraction, with well-defined mechanisms for creating and destroying connections and for detecting errors. There are about 20 calls in the API including `send()`, `recv()`, `accept()`, `connect()`, `bind()`, `select()`, `socket()`, and `close()`. Traditionally, stream sockets are implemented on top of UDP so that applications can run across any networks using TCP/IP protocol.

Our stream sockets implementation is for system area networks and was originally implemented on top of *VMMC* for PC clusters using Linux.

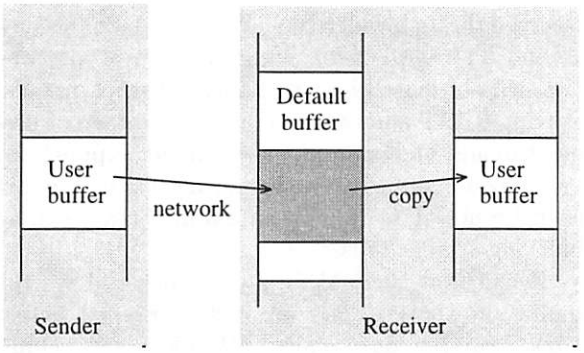
The *VMMC* model now supports a mechanism called *transfer redirection*. The basic idea is to use a default, *redirectable* receive buffer in case when a sender does not know the final receive buffer addresses. Stream sockets makes heavy use this mechanism.

Redirection is a local operation affecting only the receiving process. The sender does not have to be aware of a redirection and always sends data to the default buffer. When the data arrives at the receive side, the redirection mechanism checks to see whether a redirection address has been posted. If no redirection address has been posted, the data will be moved to the default buffer. Later, when the receiver posts the receive buffer address, the data will be copied from the default buffer to the receive buffer, as shown in Figure 3(a).

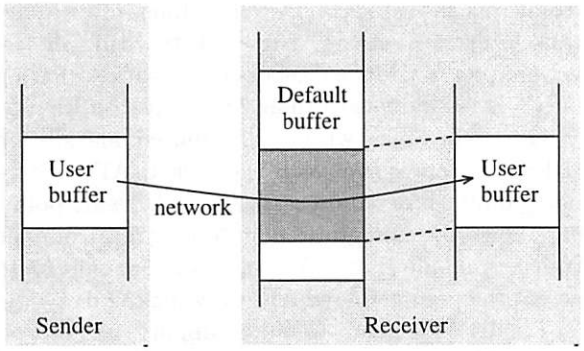
If the receiver posts its buffer address before the message arrives, the message will be put into the user buffer directly from the network without any copying, as shown in Figure 3(b). If the receiver posts its buffer address during message arrival, the message will be partially placed in the default buffer and partially placed in the posted receive buffer. The redirection mechanism tells the receiver exactly how much and what part of a message is redirected. When partial redirection occurs, this information allows the receiver to copy the part of the message that is placed in the default buffer.

VMMC stream sockets performs quite well with a one-way latency of 20 μs and a peak bandwidth of over 84 Mbytes/s. The bandwidth performance is due, in large part, to redirection. Without redirection bandwidth would be limited by the system copy bandwidth because one copy would be required by the receiver to move incoming data to its final destination.

Because our stream sockets is a user-level library, it does not allow open sockets to be preserved across `fork()` and `exec()` calls. With `fork`, the problem is arbitrating socket access between the two resulting processes. `exec` is difficult because VMMC communicates through memory and `exec`



(a) A copy takes place when the receiver posts its buffer address too late.



(b) A transfer redirection moves data directly from network to the user buffer.

Figure 3: Transfer redirection uses a default buffer to hold data in case receiver posts no buffer address or posts it too late, and moves data directly from the network to the user buffer if the receiver posts its buffer address before the data arrives.

allocates a new memory space for the process. This limitation was not due to the sockets implementation but rather it is a fundamental problem with user-level communications in general[1].

4.2 Implementing NT VMMC Sockets

Windows sockets, or WinSock, adapts the sockets communication model to the Windows programming environment. As a result, WinSock contains many Windows-specific extension functions in addition to the core Unix stream socket calls.

Instead of implementing Winsock, we decided to support only a subset of Winsock calls, the same set of functions that were in our VMMC sockets Linux implementation. The current implementation does not support (for now) out-of-band (OOB) data, scatter/gather operations, and polling with `recv()` (MSG_PEEK). We call this implementation NT VMMC Sockets.

The two main issues in implementing NT VMMC sockets were: (i) seamless *user-level* integration of the library for binary compatibility with existing applications, and (ii) integration of user-level VMMC sockets with NT kernel sockets. The solution that satisfied both requirements was using a wrapper DLL (dynamic-link library) that intercepts WinSock calls and allowed for a user-level implementation of sockets.

NT provides support for sockets via two DLLs, `wsock32.dll` (WinSock 1.1) and `ws2_32.dll` (WinSock 2.0). All calls in `wsock32.dll` end up calling into either `ws2_32.dll` or `mswsock.dll`. Since we only support a subset of the original Berkeley sockets functions that are in WinSock 1.1, we just need to deal with the functions in `wsock32.dll`. Our wrapper DLL for `wsock32.dll`, intercepts WinSock 1.1 calls and implements user-level sockets using VMMC. To disambiguate between the two identically named files we refer to the VMMC-based library as $\texttt{wsock32.dll}_{\text{vmmc}}$. By simply placing a copy of $\texttt{wsock32.dll}_{\text{vmmc}}$ in the application's directory (or path) WinSock 1.1 calls are automatically intercepted. Removing (or renaming it) allows the application to use NT's `wsock32.dll`.

We also wanted to support both *types* of stream connections: user-level VMMC and NT kernel (i.e. Ethernet). In order to accomplish this, the user-level sockets library allocates a socket descriptor table that contains one entry for each open socket, regardless of the socket type. When NT kernel sockets are used, $\texttt{wsock32.dll}_{\text{vmmc}}$ forwards calls through to either `ws2_32.dll` or `mswsock.dll`, while still maintaining a descriptor table entry. Also, our library uses calls to `wsock32.dll` in order to bootstrap the VMMC connection. Figure 4 illustrates the software layers in our user-level implementation.

Building a wrapper DLL $\texttt{wsock32.dll}_{\text{vmmc}}$ was straightforward. We used the `dumpbin` utility provided by Microsoft Win32 SDK to produce the list of exported functions in the `wsock32.dll` as well as their forwarding information. We wrote a perl script to find the function prototype for each exported function from a list of header files and produce a stub call that uses `GetProcAddress()` to obtain the address of the corresponding function in either `ws2_32.dll` or `mswsock.dll`. This script also produces the definition (.def) file needed to build the DLL along with the correct function ordinals. $\texttt{wsock32.dll}_{\text{vmmc}}$ uses `LoadLibrary()` to load both `ws2_32.dll` and `mswsock.dll` so that it

[1] Maeda and Bershad [19] discuss how to implement `fork()` and `exec()` correctly in the presence of user-level networking software.

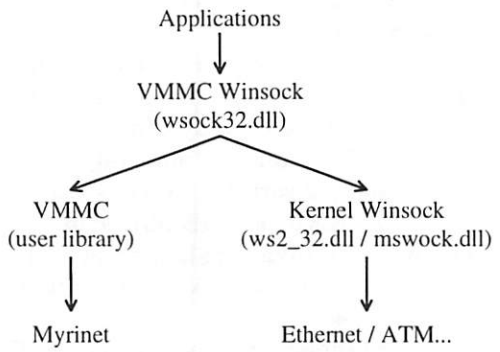


Figure 4: Socket layers

can, as needed, forward calls to them.

4.3 Lessons Learned

It turned out that we were very lucky: A staff researcher at Microsoft later pointed to us that our approach to produce a wrapper DLL worked because `wsock32.dll` is not a *preloaded* DLL. Preloaded DLL is a Microsoft term for DLLs that are loaded during system startup. They cannot be replaced by any wrapper DLLs.

However, one can easily disable the preload attribute of a DLL by removing its name from a registry (KnownDLL). Using this trick, we were able to build a wrapper around `kernel32.dll` to build a user-level fast file system. Working with DLLs gave us the flexibility to leverage user-level communication and minimize the overhead of kernel calls.

The Linux version of VMMC sockets maintains an internal socket descriptor table that is index with the socket descriptor. But WinSock adds a new data type, `SOCKET`, because socket descriptors are no longer equivalent to file descriptors, as in Linux. This change forces $\texttt{wsock32.dll}_{\text{vmmc}}$ to convert `SOCKET` values to table indices and vice versa. A small but important detail.

VMMC sockets is thread-safe and written with support for a preemptive user-level threads package for Linux [16]. When VMMC sockets was being developed, support for kernel threads in Linux was not dependable and was still work in-progress. Therefore, we decided to use a user-level threads package instead. Using user-level threads required that we worry about invoking system calls and possibly blocking the process. We added functionality to the VMMC sockets implementation to stop this from happening. VMMC sockets also integrated support for asynchronous I/O. A special socket was reserved through which requests were made to asynchronous I/O. All of this additional complexity was necessary because Linux then lacked kernel threads. Porting to NT allowed us to take advantage of kernel threads and simplify the thread support for $\texttt{wsock32.dll}_{\text{vmmc}}$. We no longer needed to support asynchronous I/O or or worry about threads calling blocking system calls.

Recall that the VMMC Linux sockets have the limitation that sockets are not preserved across `fork()` and `exec()` calls. Windows NT eliminates this limitation because it does not support the `fork/exec` model for process creation. But NT introduces a the same problem in a different form because processes can ask share sockets. Still, NT is a net gain for VMMC sockets because socket sharing in NT is less pervasive than `fork/exec` on Linux.

Our experience with a distributed file system (DFS) [22] led us to extend the sockets API to support pointer-based data transfers [13]. Using pointers allows the DFS to eliminate copying from the transfers of file data. We use `ioctlsocket()` to access the pointer-based API of $\texttt{wsock32.dll}_{\text{vmmc}}$.

Finally, we plan to add support for out-of-band (OOB) data, scatter/gather operations, and polling with `recv()` (MSG_PEEK). Further we want to produce a more complete implementation of VMMC sockets that supports many WinSock 2.0 calls directly.

5 Performance evaluation

VMMC minimizes OS involvement in common communication datapaths. However, some OS involvement is still necessary for tasks like communication setup and virtual-to-physical address translation. Therefore, we want to evaluate the impact of the operating system on the three distinct aspects of user-level communication:

- OS impact on the communication setup performance
- OS impact on the data transfer performance
- OS impact on the control transfer performance

Methodology The ideal approach would be to run identical applications on both Linux and Windows NT platforms. Unfortunately, we do not have such applications for a variety of reasons. First, our OpenGL applications are written for the NT platform because it is the single most popular platform that all high-end graphics accelerator vendors support. Second, the shared-memory applications

currently use an SVM library that was solely developed for the Win32 platform, primarily due to its mature support for kernel threads. The incompatibility between the Linux and Win32 API also made it very difficult to have a single SVM protocol library that works for both OSes.

Since we do not have the identical applications on the two platforms, we perform the evaluation as follows: First, we use microbenchmarks to measure the various aspects of the communication system and the relevant aspects of operating systems on the two platforms. Then we instrument the applications to measure the frequency of occurrence of the various events like UTLB misses and notifications on applications running on windows NT platform. Finally, we combine the two measurements to predict the impact of the OS on the applications performance.

Platform All measurements are performed on a cluster of PCs. Each has a 450 MHz Intel Pentium II processor, 1 GB memory and a Myricom network interface card with LANai 4.x microprocessor and 1MB on-board SRAM. The PCs are set up for dualboot between Linux 2.0.36 kernel[2] and Windows NT 4.0 SP5.

5.1 Impact on Communication Setup

The VMMC communication is set up via *export* and *import* calls. A process can *export* the read and/or write rights on its virtual memory buffer. Another process on a different or the same machine can gain the access rights through an import operation; afterwards, it can directly read from or write to this buffer. The export-import semantics provides protection in VMMC. Export is implemented inside the device driver and invoked through the *ioctl* mechanism. Table 1 lists the cost of kernel entry and exit using the ioctl mechanism.

Platform	Linux	Windows NT
ioctl kernel entry	1.1 μs	6.6 μs
ioctl kernel exit	1.0 μs	7.8 μs

Table 1: Overhead of kernel calls

The export call pins the exported buffer in physical memory. In current implementation, exported pages are never unpinned. The cost of pinning a user buffer is listed in Table 2. On NT, exporting a one-page buffer takes 43 μs of which 18.3 μs is

[2]We can not simply use the newer Linux 2.2.x kernel, because bringing our VMMC software up to the 2.2 kernel requires non-trival changes.

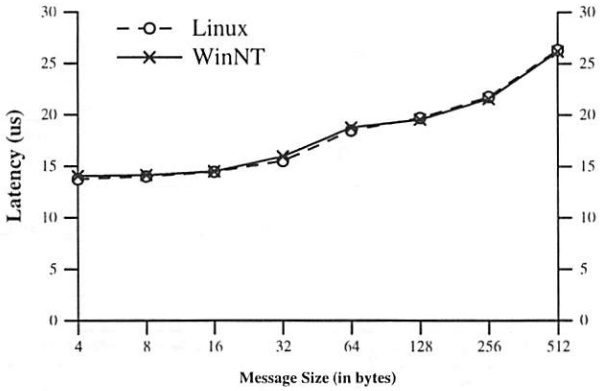


Figure 5: Remote send latency for small messages

OS specific (ioctl call + pinning a page) while importing a memory buffer costs 68 μs none of which involves the OS.

In most of our applications, the connection setup occurs only at the beginning. For client-server applications that use sockets, the connection setup happens each time a socket is opened. In both cases, the setup cost accounts for a small fraction of the execution time. The small difference in the setup performance on the two platforms should have little impact on application performance.

5.2 Impact on Data Transfer

The impact of operating system on the performance of user-level data transfer is quite limited. In VMMC, the OS involvement is needed only when the address translation is not available in the UTLB translation table. To evaluate this effect, we first measure the performance in the common case in which all the address translation is in the UTLB translation table and therefore requires no operating system involvement. We then measure the UTLB overheads when the translations are not available in the translation table. Finally we measure the UTLB overheads in a few real applications and parallel programs.

Common case Figure 5 shows the one way latency while Figure 6 shows the bandwidth of synchronous remote send operations on the two platforms. The latency is measured using a pingpong benchmark in which two processes on two different machine send synchronous messages to each other back and forth. The bandwidth is measured using a pair of processes on two different machines where one process continuously does synchronous sends to the second process. Because the PCI bus bandwidth(133 MB/s) and Myrinet bandwidth(160 MB/s) are very high, the dominant portion of time

No. Of Pages		1	4	8	12	16
Linux	Pin	3.6 μs	4.0 μs	5.0 μs	6.4 μs	10.7 μs
	Unpin	2.8 μs	11.0 μs	19.1 μs	28.2 μs	34.5 μs
Windows NT	Pin	2.9 μs	8.9 μs	17.0 μs	24.0 μs	32.8 μs
	Unpin	1.6 μs	4.6 μs	8.6 μs	12.5 μs	16.7 μs

Table 2: Overhead of pinning and unpinning pages in memory (excluding ioctl overhead).

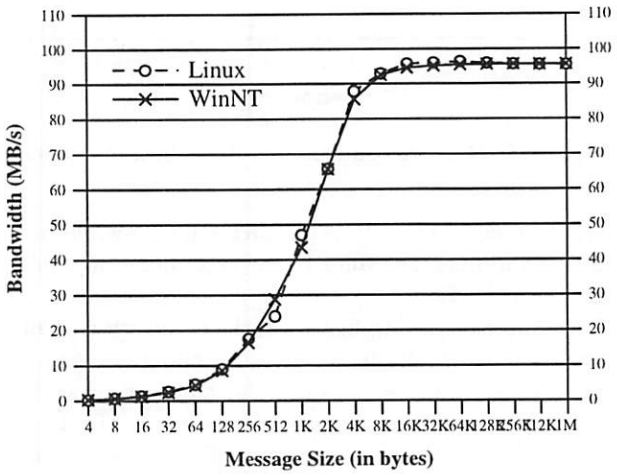


Figure 6: One-way send bandwidth

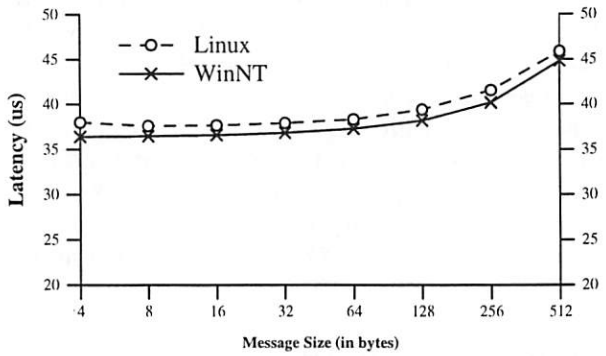


Figure 7: Remote fetch latency

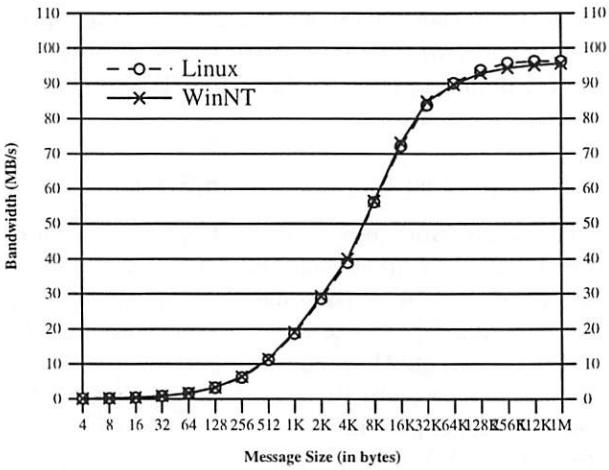


Figure 8: Remote fetch bandwidth

for small message communication is the processing overhead on the the network interface, therefore is independent of the OS. The one-way latency for small messages is 14 μs while the peak bandwidth is about 96 MB/s on both platforms.

The VMMC communication model also provides a remote-read capability to fetch data from the memory of the remote machine without the involvement of host processor that machine. Figures 7 and 8 show the latency and bandwidth performance respectively. Both measurements are performed using a pair of processes on two different machines where one process repeatedly performs reads from the second process' memory using the remote-read operation.

As expected, we see that the OS has little impact on the data transfer in the common case. Later optimizations on NT further reduce the latency by 6 μs.

UTLB overheads Two kinds of overhead occur in the slow path but not the fast path. The first overhead occurs when there is a *host translation miss*: when the local source buffer of a send operation or the local destination buffer of a fetch operation is not currently pinned and needs to be pinned for DMA transfer. Since only a limited amount of virtual memory can be pinned, this may result in unpinning other pages. The second overhead is caused by a translation miss in the network interface UTLB cache; in this case, the network interface directly DMAs the translation from the host memory. Note that only the first overhead involves the operating system (Table 1 for ioctl overhead and Table 2 for pinning and unpinning overhead). To evaluate the overhead of UTLB operations in a more realistic scenario, we instrumented 7 applications that run either directly on VMMC NT or on VMMC Winsock and report the measured UTLB overhead.

Applications From the SPLASH-2 suite, we chose 4 SVM applications, *radix*, *barnes*, *fft*, and *water-*

Applications	vis	glaze	vdd	fft	barnes	radix	water
memory footprint	21.4 MB	39.9 MB	78.9 MB	57.4 MB	16.6 MB	54.4 MB	8.9 MB
host lookup hit rate	99.93%	99.99%	99.99%	79.38%	98.62%	67.36%	99.35%
avg lookup hit cost	0.65 μs	0.63 μs	2.41 μs	0.32 μs	0.46 μs	0.32 μs	0.40 μs
avg lookup pages	1.00	1.94	54.8	1.00	1.00	1.04	1.00
avg pin overhead on NT	54.3 μs	401.6 μs	939.0 μs	47.3 μs	53.1 μs	50.9 μs	55.6 μs
avg pages pinned on NT	1.00	97.0	289.0	1.00 μs	1.13 μs	1.00 μs	1.25 μs
avg miss overhead on NT	63.4 μs	557.1 μs	1294 μs	50.7 μs	57.5 μs	54.6 μs	60.0 μs
predicted ovhd on Linux	41.7 μs	410.1 μs	882 μs	34.7 μs	40.5 μs	42.0 μs	47.4 μs
avg host overhead on NT	0.69 μs	0.69 μs	2.54 μs	10.71μs	1.25 μs	18.04μs	0.79 μs
predicted ovhd on Linux	0.68 μs	0.67 μs	2.50 μs	7.41μs	1.01 μs	13.92μs	0.71 μs

Table 3: UTLB performance without memory constraint

Applications	fft	barnes	radix	water
memory footprint	57.4 MB	16.6 MB	54.4 MB	8.9 MB
memory constraint	50.0 MB	16.0 MB	30.0 MB	6.0 MB
host lookup hit rate	64.46%	96.67%	54.99%	97.63%
host pin-page rate	35.54%	3.33%	45.01%	2.37%
host unpin-page rate	23.36%	2.13%	36.10%	2.27%
avg lookup hit cost	0.36 μs	0.50 μs	0.35 μs	0.40 μs
avg pin overhead on NT	39.1 μs	40.0 μs	42.1 μs	29.6 μs
avg pages per pin on NT	1.00	1.07	1.00	1.07
avg unpin overhead on NT	33.2 μs	36.7 μs	35.2 μs	35.0 μs
avg miss overhead on NT	42.2 μs	49.6 μs	68.7 μs	32.8 μs
predicted ovhd on Linux	21.6 μs	29.3 μs	46.4 μs	8.6 μs
avg host overhead on NT	15.23 μs	2.13 μs	31.1 μs	1.27 μs
predicted ovhd on Linux	7.91 μs	1.45 μs	21.1 μs	0.59 μs

Table 4: UTLB performance with memory constraint

nsquare. They all use the SVM protocol developed by our colleagues at Princeton [26]. The SVM protocol communication is built directly on top of VMMC, using the send and remote fetch mechanism. In addition, we selected 3 applications from Princeton Display Wall Project [11] *glaze*, *isosurface*, and *vdd*. These applications use VMMC Winsock as their underlying communication mechanism.

Vis is a visualization program, implemented by our colleagues that uses a cluster of 13 PCs to visualize isosurfaces of scientific data. The program has three components: client control, isosurface extraction, and rendering. The client control runs on a single PC to implement the user interface for users to steer the entire visualization process. The isosurface extraction uses 4 PCs in the cluster to extract isosurfaces in parallel, and sends them to the rendering program which runs on 8 PCs. Each renderer PC simply receives isosurfaces, renders and displays the rendered images on a tiled, scalable display wall. VMMC Winsock is used for all interprocess communication among the PCs running the client control, isosurface extraction, and rendering programs.

Glaze is a commercial OpenGL evaluation program from Evans & Sutherlands. An OpenGL framework, developed by our colleagues, allows us to use one PC to drive OpenGL animation of any commercial software on a tiled display. We ran the glaze program on a PC; a custom wrapper DLL (opengl32.dll) intercepts the OpenGL calls made by the program and distributes the commands to 8 separate PCs which together drive a 2x4 tiled display. The wrapper DLL transmits rendering commands over the VMMC Winsock layter to the renderers. The renders behave just like those in *vis*: they simply receive the commands and render them for its portion of the tiled display.

Vdd is a virtual display driver that implements a large desktop that has the same resolution (3850x1500) as the Display Wall. It is installed on a 450 MHz Pentium II PC running NT 5B2 OS. A user process is responsible for packetizing the

updates made to the vdd's memory-resident framebuffer and distributing them to the 8 PCs that drive the wall. We used NT VMMC Sockets with the pointer-based extensions to distribute framebuffer updates efficiently [13].

Results Table 3 presents OS-dependent components of UTLB overhead, without any memory constraint. The miss rates and overheads for the NT platform are measured directly by running the 7 applications on our NT VMMC cluster. We use these numbers, as well as the micro-benchmark results, to predict the overheads on Linux. In the table, *host pin rate* is the ratio between the number of pin operations and the number of UTLB lookups; and similarly for *host unpin rate*. *Memory footprint* is the total amount of distinct virtual memory that is involved in data transfer for each application. Lower page-pin rate (or lookup miss rate) translates into lower average UTLB overhead. Since the host UTLB miss rates are quite low, the OS overhead for pinning and unpinning user buffers has little impact on the average UTLB host overhead.

Under tight memory constraints, such as in a multi-programming environment or a large-memory applications, there may not be enough physical memory to absorb the entire memory footprint of applications. UTLB deals with such low-memory situations by unpinning virtual pages that are not currently involved in data transfer. This is a critical feature for UTLB-based VMMC to be useful in a multi-programming environment. Table 4 presents the the same experiments with additional memory constraints such that the total amount of application-pinnable physical memory is less than the memory footprint. Due to varying nature of the applications, the memory constraint is set differently on a per-application basis. And also, it turned out that our Display Wall applications require most of their working sets to be pinned as receive buffers. This is because each DisplayWall application uses a small buffer (often less than 1 MB) to gather commands and send the whole chunk to the receivers. Therefore, we only present the memory-constrained results for the 4 SVM applications. Again, the measurements are all taken by running the applications on the NT cluster, and predictions are made for the Linux platform.

With low memory constraint, we see higher UTLB miss rates and page unpin rates. Note that our prediction for UTLB performance on Linux suggests that faster page-pin and page-unpin OS calls further reduce the average UTLB overhead by as much as 50%. We conclude that improving system calls such as page-pin and page-unpin benefits user-level communication.

5.3 Impact on Control Transfer

In VMMC, messages are deposited directly into the memory of the receiving process without interrupting the host processor. The receiving process can detect message arrival by polling memory locations. But it can also use the VMMC notification mechanism to reduce the CPU polling overhead. For instance, the sockets library uses notifications extensively to avoid polling while waiting for data to arrive on a socket.

The VMMC notification mechanism allows a receiving process to associate a message handler with each exported buffer. The message handler is executed when a message with notification request arrives in the exported buffer. Since interrupts are used to deliver notifications to the corresponding process, different OS platforms show different performance.

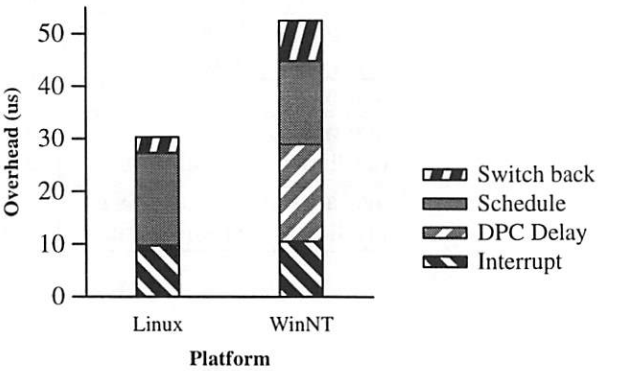


Figure 9: Receive-side notification overhead breakdowns

Figure 9 shows the breakdown of the notification overhead on the receiving machine. The interrupt cost is the time from when the network interface issues the interrupt to the time when the interrupt handler in the driver starts to execute. Surprisingly[3], both Linux and NT take about 10 μs to reach the handler on our test platform. Once interrupted, Linux uses 17 μs to schedule the user level message handler through the process signal mechanism. On NT, because there are restrictions on what can be executed in the interrupt handler, the interrupt handler has to queue a Deferred Procedure Call (DPC) in the kernel which invokes the user-level notification handler later. By the time the control reaches the start of the DPC handler,

[3]The cost on some other similar machines is much less. We are still investigating the cause for this high overhead. This won't affect our platform comparison.

18 μs has already passed. After this, NT uses 16 μs to schedule the user level message handler by waking up the user thread that executes the message handler. Once the notification handler terminates, Linux uses 3 μs to return to the interrupted process while NT uses 7 μs to switch back to the interrupted thread. The overall overhead for a notification is 30 μs on Linux and 52 μs on NT.

6 Related Work

User-level communication architectures for system-area networks is a very active research area [15, 8, 12, 23, 20, 2, 24, 10].

But, only two other user-level communication systems are currently available for Windows NT. A Myrinet-based Fast Messages (FM) [20] implementation has recently been ported to the NT platform. Since it is implemented entirely at user-level, it does not require any OS support. U-net [2] has also been ported to an NT cluster. However, they use Fast Ethernet and require kernel modifications to implement low-overhead kernel entry and exit.

While this paper only presents the results of our porting VMMC to Windows NT, there is a lot of published work describing the various efforts that are part of the SHRIMP project. The VMMC mechanism [5, 15, 9], our low-level communication layer, performs direct data transfers between virtual memory address spaces bypassing the operating system. We have also designed and implemented several compatibility communication software including NX message-passing library [1], RPC [3], and Unix stream sockets [14], and showed that they deliver good performance. Finally, applications are implemented using the higher level APIs: (i) distributed file system [22], (ii) SPLASH2 [25], and (iii) distributed OpenGL graphics applications.

7 Conclusions

Our experience with porting to Windows NT has been positive. NT has fairly complete device driver support and good documentation. In addition, it has good support for threads and SMP systems. We found the DLL mechanism to be very convenient. The Display Wall project uses a Windows NT-based VMMC cluster for high-performance communication.

Our measurements indicate that the OS overhead on Windows NT is significantly higher than on Linux. However, it provides much better functionality especially in terms of threads and SMP systems.

Finally, we find that the VMMC user level communication architecture is successful in delivering high-performance to applications on both platforms.

Our software is publicly available at `http://www.cs.princeton.edu/SHRIMP/`.

Acknowledgments

This project is sponsored in part by DARPA under grant N00014-95-1-1144, by NSF under grant MIP 9420653 and CDA96-24099, and by Intel Corporation.

We would like to thank Paul Pierce from Intel Corp. for doing the initial port of VMMC to NT, Allison Klein and Rudro Samanta from Princeton University for providing us with Display Wall applications, and Richard Shupak from Microsoft Corp. for answering our NT questions. We also want to thank the program committee, anonymous reviewers, and the shepherd of this paper, Thorsten von Eicken, for the helpful feedback.

References

[1] Richard Alpert, Cezary Dubnicki, Edward W. Felten, and Kai Li. Design and Implementation of NX Message Passing Using SHRIMP Virtual Memory-Mapped Communication. In *Proceedings of the International Conference on Parallel Processing*, August 1996.

[2] Anindya Basu, Matt Welsh, and Thorsten von Eicken. Incorporating Memory Management into User-Level Network Interfaces. In *Presentation at IEEE Hot Interconnects V*, August 1997. Also available as Tech Report TR97-1620; Computer Science Department, Cornell University.

[3] A. Bilas and E. W. Felten. Fast RPC on the SHRIMP Virtual Memory Mapped Network Interface. *Journal of Parallel and Distributed Computing*, 40(1):138–146, January 1997.

[4] Andrew D. Birrell and Bruce Jay Nelson. Implementing Remote Procedure Calls. *ACM Trans. Comp. Sys.*, 2(1):39–59, November 1984.

[5] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. A Virtual Memory Mapped Network Interface for the Shrimp Multicomputer. In *Proceedings of the 21st Annual Symposium on Computer Architecture*, pages 142–153, April 1994.

[6] Matthias A. Blumrich, Richard D. Alpert, Yuqun Chen, Douglas W. Clark, Stefanos N. Damianakis, Cezary Dubnicki, Edward W. Felten, Liviu Iftode, Kai Li, Margaret Martonosi, and Robert A. Shillner. Design Decisions in the SHRIMP System: An

Empirical Study. In *Proceedings of the 25th Annual Symposium on Computer Architecture*, June 1998. To appear.

[7] Matthias A. Blumrich, Cezary Dubnick, Edward W. Felten, and Kai Li. Protected, User-Level DMA for the SHRIMP Network Interface. In *IEEE 2nd International Symposium on High-Performance Computer Architecture*, pages 154–165, February 1996.

[8] Greg Buzzard, David Jacobson, Milon Mackey, Scott Marovich, and John Wilkes. An Implementation of the Hamlyn Sender-Managed Interface Architecture. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 245–260, October 1996.

[9] Yuqun Chen, Angelos Bilas, Stefanos N. Damianakis, Czarek Dubnicki, and Kai Li. UTLB: A Mechanism for Translations on Network Interface. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 193–204, October 1998.

[10] Compaq/Intel/Microsoft. *Virtual Interface Architecture Specification, Version 1.0*, December 1997.

[11] Princeton University Computer Science Department. The Princetion Display Wall Project. http://www.cs.princeton.edu/omniwall, 1999.

[12] C. Dalton, G.Watson, D. Banks, C. Calamvokis, A. Edwards, and J. Lumley. Afterburner. *IEEE Network*, 7(4):36–43, 1995.

[13] Stefanos N. Damianakis. *Efficient Connection-Oriented Communication on High-Performance Networks.* PhD thesis, Dept. of Computer Science, Princeton University, May 1998. Available as technical report TR-582-98.

[14] Stefanos N. Damianakis, Cezary Dubnicki, and Edward W. Felten. Stream Sockets on SHRIMP. In *Proc. of 1st Intl. Workshop on Communication and Architectural Support for Network-Based Parallel Computing (Proceedings available as Lecture Notes in Computer Science 1199)*, February 1997.

[15] Cezary Dubnicki, Angelos Bilas, Kai Li, and James Philbin. Design and Implementation of Virtual Memory-Mapped Communication on Myrinet. In *Proceedings of the IEEE 11th International Parallel Processing Symposium*, April 1997.

[16] David R. Hanson. *C Interfaces and Implementations: Techniques for Creating Reusable Software.* Addison-Wesley, Reading, Massachusetts, 1997.

[17] Liviu Iftode, Cezary Dubnicki, Edward Felten, and Kai Li. Improving Release-Consistent Shared Virtual Memory using Automatic Update. In *Proceedings of IEEE 2nd International Symposium on High-Performance Computer Architecture*, February 1996.

[18] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.3BSD Unix Operating System.* Addison Wesley, 1989.

[19] Chris Maeda and Brian N. Bershad. Protocol Service Decomposition for High-Performance Networking. In *Proceedings of 14th ACM Symposium on Operating Systems Principles*, pages 244–255, December 1993.

[20] Scott Pakin, Mario Lauria, and Andrew Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of Supercomputing '95*, 1995.

[21] R. Samanta, A. Bilas, L. Iftode, and J.P. Singh. Home-based SVM protocols for SMP clusters: Design and Performance. In *Proceedings of 4th International Symposium on High-Performance Computer Architecture*, February 1998.

[22] Robert A. Shillner and Edward W. Felten. Simplifying Distributed File Systems Using a Shared Logical Disk. Technical Report TR-524-96, Princeton University Computer Science Department, Princeton NJ, 1996.

[23] J.M. Smith and C.B.S. Traw. Giving Applications Access to Gb/s Networking. *IEEE Network*, 7(4):44–52, July 1993.

[24] H. Tezuka, A. Hori, and Y. Ishikawa. PM: a High-Performance Communication Library for Multi-user Parallel Environments. Submitted to Usenix'97, 1996.

[25] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. Methodological Considerations and Characterization of the SPLASH-2 Parallel Application Suite. In *Proceedings of the 22nd Annual Symposium on Computer Architecture*, May 1995.

[26] Yuanyuan Zhou, Liviu Iftode, and Kai Li. Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 75–88, October 1996.

Windows NT in a ccNUMA System

B. Brock, G. Carpenter, E. Chiprout, E. Elnozahy, M. Dean, D. Glasco,
J. Peterson, R. Rajamony, F. Rawson, R. Rockhold, A. Zimmerman

IBM Austin Research Laboratory
Bldg. 904, MS-9460, 11501 Burnet Road
Austin, TX 78758
mootaz@us.ibm.com

Abstract

We have built a 16-way, ccNUMA multiprocessor prototype to study the feasibility of building large scale servers out of *Standard High Volume* (SHV) components. Using a cache-coherent interconnect, our prototype combines four 4-processor SMPs built using 350MHz Intel Xeon™ processors, yielding a 16-way system with a total of 4 GBytes of physical memory distributed over the nodes. Such an environment poses several performance challenges to Windows NT®, which assumes that memory is equidistant to all processors. To overcome these problems, we have implemented an abstraction called a *Resource Set*, which allows threads to specify their execution and memory affinity across the ccNUMA complex.

We used a suite of parallel applications to evaluate the scalability and performance of the system. Our results confirm the feasibility of building ccNUMA systems out of SHV components, and suggest that memory allocation affinity should be incorporated as part of the standard Windows NT API. Also, the performance degradation due to poor bus bandwidth in the current generation of Intel-based processors often dominates the degradation due to the latency of remote memory accesses.

1. Introduction

There is an increasing need for powerful servers to meet the processing demands of modern distributed systems, transaction processing systems, and Internet data providers. Traditional server systems use proprietary mainframe computers or other large computer systems that are powerful and robust, though expensive. Recently, there has been an increasing demand for servers built using commodity, off-the-shelf processors and components. In particular, symmetric-multiprocessor systems (SMP) that use Intel's x86 processors and run Windows NT are in increasing favor due to their low cost, application software availability, and success in the personal computer and workstation markets. However, physical limits impose restrictions on the size of SMP systems, and most SMP systems based on the Intel Pentium® II processor contain at most 4 processors.

In order to investigate the feasibility of using these SHV components to build larger servers, we have used a cache-coherent interconnect to connect four 4-processor SMP nodes. Each node contains 1 GByte of main memory and four 350MHz Intel Xeon processors. The resulting system is a cache-coherent, non-uniform memory access (ccNUMA) machine, which poses several performance challenges to Windows NT since it is written to assume that all of system memory is equidistant to the processors. The focus of this paper is our addition to Windows NT to support memory allocation affinity.

Our enhancement to Windows NT 4.0 includes extending the Basic Input Output System (BIOS) and Hardware Abstraction Layer (HAL) to present the operating system with a single system image. These extensions do not require any modifications to the NT source code (to which we did not have access), and allow Windows NT to treat the system as a single 16-way SMP. The second major component of our enhancement is an implementation of a *Resource Set* abstraction (RSet), which allows application programs to control resource allocation in order to improve performance. The current implementation of RSets consists of a collection of Application Program Interfaces (APIs), Dynamic Link Libraries (DLLs), and a kernel-mode device driver that allows applications to control where memory is allocated.

We used RSets to tune a suite of six parallel programs, and studied the scalability of the applications under different system configurations. Without affinity in memory allocation, several

applications scale poorly as the system's size increases. This result demonstrates that adding support for memory affinity to Windows NT's standard Application Programming Interface (API) and tuning the application to benefit from this extended API are necessary for ccNUMA systems such as ours. However, such tuning may not be necessary if the machine were to include a remote memory cache and larger processor caches.

Our results also suggest that the poor bus bandwidth (100MHz) of the current generation of Intel-based SMPs often has more of a detrimental effect on performance than the latency of accessing remote memory across the interconnect. This result is somewhat surprising since one would expect the latency of the NUMA interconnect to be the major source of overhead in a ccNUMA system.

Finally, applications that scale well on an SMP were found to continue to scale reasonably as the system size was increased, until the configuration parameters (bus bandwidth, remote memory latency, and L2 cache sizes) prevented any further scaling. On the other hand, applications with poor scalability trends on small systems were predictably unable to scale better on larger ones.

The remainder of the paper is organized as follows. Section 2 reviews previous work. Section 3 gives an overview of our implementation effort. Section 4 describes resource sets. Section 5 presents the results of the experimental evaluation. Our conclusions are presented in Section 6.

2. Previous Work

Over the years several research ccNUMA machines have been constructed, such as Alewife [Ale], DASH [Len] and FLASH [Flash]. These systems have been built either by constructing processor and memory nodes from scratch, or by combining pre-existing hardware using a combination of hardware modifications and interconnection fabrics. Recent years have seen the introduction of several affordable, general purpose ccNUMA and scalable shared-memory implementations such as the Silicon Graphics Origin™ [SGI], the Sequent NUMA-Q™ [Sting], the Data General Aviion™ NUMALiiNE™ [DG] and the Unisys Cellular MultiProcessor [Uni]. Except for the Origin, all of these systems use Intel IA-32 processors. The Sequent and Data General systems are built using standard high-volume 4-processor SMP nodes. The Origin uses the R10000™ processor and unique memory and I/O structures within as well as between 2-processor nodes. Unisys' Cellular MP also uses a unique arrangement of 2-processor nodes and claims uniform access times to memory. Our hardware prototype is based on an extension of the Fujitsu Synfinity™ interconnect used in Fujitsu's *team*server™ [FJST].

Until recently, most of the work done on ccNUMA systems used a variant of UNIX as the operating system. For instance, SGI Irix™ 6.4 (Cellular Irix™) supports the ccNUMA features of the Origin while Sequent has historically had its own implementation of UNIX known as Dynix/ptx®. To the best of our knowledge, Microsoft's direction for scaling Windows NT[NT] beyond standard SMPs has been to provide clustering through the Microsoft Cluster Service[MSCS]. We are also not aware of any existing or planned support for ccNUMA systems in the standard release of Windows NT. However, some of the recent ccNUMA systems, including the Sequent NUMA-Q and the Fujitsu *team*server, have supported NT either as an alternative to UNIX, or as the primary operating system provided on the system. Our NT work is based on the Fujitsu implementation for two-node systems.

Although the Splash [Spl] and Splash-2 [Spl2] benchmark suites have been widely used in the academic community to measure multiprocessor performance [Shrimp, Len], most of the commercial server vendors have been more concerned about reporting performance data using the Transaction Processing Performance Council's TPC-C and TPC-D benchmarks [Gray, TPC]. However, while the TPC benchmarks are good indicators of overall transaction processing performance, they require a large investment to set up and execute properly. For this reason, we have chosen to study a subset of Splash-2 and other scientific workloads. With the exception of a few studies on distributed shared memory systems [Brazos, Sch], most of the Splash-2 studies reported to date have been carried out on UNIX systems.

The value of affinity scheduling has been recognized for SMP machines where many systems attempt to run a thread on the same processor that it ran on last in the hope of reusing data that is already in the cache [Vas]. Our affinity implementation may be viewed as an extension of this notion by colocating threads with the physical memory that they access. Although our implementation is indirect (since we do not have NT source code access), it should be relatively straightforward for Microsoft to implement similar functions directly in the NT executive.

The work of the FLASH project on improving data locality for ccNUMA machines used page

migration and replication rather than affinity allocation to improve memory reference locality [Ver]. The Silicon Graphics Origin adds specialized hardware to support an efficient implementation of page migration. The Sequent, Data General and Unisys implementations all split the machine into communicating partitions, each of which runs a distinct copy of the operating system.

3. System Overview

3.1 Hardware Overview

We have constructed a 16-processor ccNUMA system by using a Synfinity interconnect switch to connect four PentiumII-based, Fujitsu *teamserver* SMP nodes. Each node contains four 350 MHz Intel Xeon processors, each with a 1MB L2 cache, 1 GByte of RAM, a standard set of I/O peripherals, and a Mesh Coherence Unit (MCU). The MCU provides coherent access to the memory and I/O devices that exist on other nodes. We designed a hardware card to attach the MCU to the Synfinity switch, which connects the four nodes together to form the 16-processor system. We configured the switch to provide 720 MB per second per link per direction in the prototype. A remote memory access is approximately 3 times slower than a local one.

The MCU in each node snoops the node's local memory bus and uses a directory-based cache coherence protocol to extend memory coherence across nodes. The MCUs exchange point-to-point messages over the switch to access remote memory and to maintain cache coherence over the entire system. The MCU defines a 4-node memory map that effectively partitions a standard 4 GByte physical address space into 4 areas of 1 GByte each, one for each of the nodes in a 4-node system. In addition to memory, memory-mapped I/O and I/O port addresses are also remapped to allow a processor to access the memory-mapped I/O and I/O ports of remote nodes.

3.2 Enabling NT on a ccNUMA System

We enhanced the BIOS and the NT Hardware Abstraction Layer (HAL) supplied by Fujitsu in order to enable Windows NT to run on the 16-processor system (the Fujitsu implementation could support a maximum of two nodes). When powered on, the system starts booting as four separate SMP systems. After the BIOS code on each node is executed, the system executes a BIOS extension (eBIOS) before booting the operating system. The eBIOS reconfigures the four SMP nodes into one 16-way ccNUMA system. Our modifications to the NT HAL support remote interprocessor interrupts, and provide access to remote I/O devices and I/O ports by remapping them as necessary. The combination of the HAL and eBIOS code presents Windows NT with a machine that appears to be a 16-processor SMP with 4 gigabytes of physical memory.

The eBIOS allows the system to be partitioned at boot time into smaller NUMA systems. For example, the eBIOS can partition the system into two 2-node systems, each with 8 processors and 2 GBytes of physical memory. Each partition runs a distinct copy of Windows NT. Other configurations for partitioning the 16-way system into separate systems are also possible. The eBIOS can also "deactivate" processors in a node at boot time allowing us to create nodes with fewer processors for configuration benchmarking purposes.

4 Supporting Memory Affinity in NT

Operating systems on SMP architectures try (when other constraints permit) to schedule threads on the same processor on which they have previously executed. Creating an affinity between a thread and its cache footprint in this manner results in good cache hit ratios, contributing to an application's performance. In addition to supporting such implicit "bindings", Windows NT also permits threads to explicitly specify the subset of processors on which they should be scheduled for execution.

If the performance of a ccNUMA system is to scale as more nodes are added, the operating system must accommodate the variability in memory access times across the system. In particular, a thread's memory allocation requests must be satisfied such that the majority of its memory accesses are served by the node on which it executes. Affinitizing memory allocations in this manner enables applications to take full advantage of the system hardware by reducing interconnect traffic. Indeed, an application may suffer in performance if most of its accesses are to memory residing on remote nodes.

Currently, Windows NT 4.0 considers all of the physical memory in a system to be equidistant to the processors. Since the physical memory frames are indistinguishable, NT does not have any mechanism for affinitizing memory allocations. We have implemented a solution that works around the performance penalties of this limitation. Our solution provides the application with an API permitting it

to exercise control over the physical memory used to satisfy explicit memory allocation requests. In our experience, we find a resulting improvement in application performance suggesting that this kind of support should become part of the standard API for Windows NT.

Our ccNUMA API is based on a Resource Set (RSet) abstraction. Intuitively, an RSet groups several resources in such a way that a thread that is bound to a resource set consumes resources exclusively from that set. For example, one could specify an RSet containing the processors and physical memory available to one node. A thread that is bound to such an RSet will execute only on processors in that node, and have its memory allocations backed only by physical memory on that node.

RSets are flexible. They can combine the resources in two different nodes, include resources spanning different nodes, contain a partial set of the resources on one node, or any other combination that suits the application needs. Furthermore, they can be manipulated using union and intersection operations and can also form hierarchies, whereby one large RSet is made to contain several smaller RSets. To simplify the interface, our library provides a global RSet that contains all resources in the system. Thus, an application can build additional RSets by specifying subsets of the global one. We have implemented the RSet abstraction using an additional HAL call (through which we find the resources available in the system) and a combination of DLLs, backed by an NT kernel-mode device driver.

The RSet implementation provides fine-grained affinity control. Functions in the API fall into the following categories:

- Determining the system configuration.
- Creating and manipulating RSets.
- Allocating virtual memory that is backed by the physical memory contained in an RSet.
- Binding processes and threads to the processors in an RSet.

We have implemented the RSet abstraction using a combination of DLLs, backed by an NT kernel-mode device driver. Furthermore, we also provide a higher level API that provides a simplified interface to the RSet abstraction similar to traditional thread packages. Thus, an application programmer can use the RSet facility indirectly through the familiar interface of a thread library, or can access it directly to exercise greater control.

4.1 Allocating Virtual Memory Based on an RSet

There is no mechanism in Windows NT to constrain the set of physical memory pages that should back a range of virtual memory addresses. We have provided an interface similar to *VirtualAlloc*(), which supports the specification of an RSet. This interface allocates *locked* virtual memory that is backed by system memory as specified by the nodes identified through **memory_rset**:

```
void* NumaVirtualAllocLocked (
        void* start_addr,
        size_t *pages,
        RSet *memory_rset);
```

Despite the lack of directed memory allocations in NT, if one can ensure that the system memory backing a range of pages satisfies our requirements, locking the pages in memory forces the mapping to remain unchanged as long as the application is active. The challenge is to coerce NT into backing the virtual pages with memory from the requested node(s). To accomplish this, we have implemented the following approach. First, the *NumaVirtualAllocLocked* routine allocates the number of pages requested, mapping the virtual addresses into the caller's address space. It then increases the working set size of the calling process by the number of pages requested and uses *VirtualLock*() to lock the range into memory. Next, it passes the address and length of the virtual memory range to our *NumaMem* device driver which returns a list of the nodes whose real memory backs each page. For each page that is not "correctly" backed, that page is modified, released, and reallocated. This process repeats until all pages are correctly backed. The modify step was added once we observed empirically that it decreased the likelihood of NT handing the same page back to us.

The *NumaMem* device driver translates a given virtual memory address to its physical address, determines the node which "owns" that physical address, and returns that node identifier to the caller. For improved efficiency, a virtual address range can be passed to the driver, and a list of node identifiers (one per virtual page) will be returned. To enable the mapping between a physical address and a node identifier (as required by the *NumaMem* driver), we have exported an interface from our HAL implementation which provides not only the memory-range-to-node-id mapping, but also the system topology information (e.g. number of nodes,

which processors are in which node, etc.).

The lack of NT source code access or an appropriate NT API dictates that we indirectly manipulate the page-table structure. As a result, the time required to set up the memory affinity mappings is large and unpredictable. Consequently, we would not advocate this implementation as a permanent solution to the lack of memory allocation affinity in Windows NT. Instead, this implementation allows us to study the potential benefits of including such support in the system. Our results suggest that this support should become an integral part of Windows NT as it moves to scale up to large system configurations.

A constraint of our approach is that we can only affinitize memory that is allocated through our API. Thus, an application must use our API instead of *malloc*() in order to allocate affinitized memory. In addition, we do not have any control over the placement of the program text, the data+BSS regions, and the individual thread stacks. These can be affinitized (or replicated in the case of program text) easily when memory affinity support is integrated within Windows NT

4.2 Page Coloring

Preliminary experiments on our prototype indicated that page coloring has a significant impact on application performance. With physically addressed caches that are not fully associative, a poor virtual to physical address space mapping can cause cache conflicts. Page coloring is a mechanism that can potentially reduce these conflicts. Each physical memory page is assigned a "color" such that similarly colored pages map into the same cache region. By cycling through the available colors when mapping contiguous virtual memory pages to physical pages, cache conflicts can be reduced when spatially close data structures are accessed. A significant advantage of page coloring is that it makes application performance predictable.

When virtual memory is committed using the *VirtualLock*() function, we discovered empirically that Windows NT backs up the virtual range with a set of physical pages that are almost perfectly colored. We wrote our *NumaMem* driver such that the pages it returns are perfectly colored. Lacking the ability to directly manipulate the page-table structure, *NumaMem* uses information about the cache size and the identity of the physical page that backs a virtual page to iterate until the pages are perfectly colored.

4.3 Affinity Policies

Using our Windows NT device driver, we have implemented two different classes of affinity policies: one for thread execution and one for memory allocation. The thread affinity policies are:

- **Float:** This is the default NT policy. Threads are eligible to run on any processor at any time. In order to maximize cache reuse, NT tries (when other constraints permit) to schedule a thread on the same processor on which it has previously executed [NT].
- **Fill:** In this policy, as many threads are bound to a node as there are processors before we continue to the next node.
- **Round Robin:** In this policy, threads are bound such that the first thread is assigned to the first node, the second thread to the second node, and so on in a round robin fashion.

We have several memory affinity policies, including:

- **Any:** The allocated virtual memory is backed by physical memory from any node in the system. This differs from the default policy only in that the virtual memory range is locked.
- **Any-striped:** This differs from **Any** in that the pages in the range are uniformly "striped" across the nodes in the system.
- **Local:** The allocated virtual memory is backed by physical memory local to the node on which the thread executes.
- **Remote:** The allocated virtual memory is backed by physical memory this is *not* local to the node on which the thread executes. This policy allows us to determine an application's sensitivity to memory affinity.

Generally, there is no single combination of these choices that yields the best performance for *all* the applications we studied. The next section presents a performance study using these allocation policies.

5 Experience

To test the effects of using the RSet abstraction to provide memory affinity support in Windows NT, we have conducted several experiments to study the performance of parallel applications on our prototype. The application suite consists of four applications from the Stanford Splash-2 benchmark suite [Spl, Spl2], a parallel program for matrix multiplication, and an implementation of a successive

over-relaxation algorithm. The six applications are:

- Ocean-contiguous: 4-D 514x514 grids
- Raytrace: balls4 scene, Resolution = 256x256
- 3DFFT: 20 iterations, 64x128x64 3D array
- Water-spatial: 5 steps on 32768 molecules
- Matrix Multiply: 1024x1024 matrices
- Jacobi: 500 iterations on a 2000x600 array

The problem sizes were chosen such that even at 16 processors, the collective caches in the system will not accommodate the entire application data set. Note however, that the applications may work for the majority of their lifetimes with a much smaller working set.

We have modified each application to make use of the RSet abstraction. The extent of the modification depended on the application. For all of the applications except matrix multiply, the desired memory affinity could be achieved by just modifying the memory allocation calls of the key data structures at the beginning of the program. For matrix multiply, the code had to be rewritten to replicate one of the multiplier matrices over the entire NUMA machine complex. In this application, the second multiplier matrix is accessed by each thread, making it ineffectual to affinitize it to any particular node. Thus, in the absence of a remote memory cache, a large portion of the memory accesses are to remote memory. Replicating the second matrix avoids the performance hit.

We executed each of the modified applications on several system configurations under each of the policies for thread and memory placement listed in Section 4.2. Each experiment was repeated several times to ensure statistical accuracy. The time we report includes the time for running the application, but it does not include the time taken for initializing the memory pools in the *NumaMem* driver. This is consistent with our goal from the performance study, which is to identify the potential benefits of including memory affinity support inside Windows NT. The cost of initialization of the memory pools in the *NumaMem* driver is not relevant to this goal, since we are not advocating this as a technique for memory allocation anyway. Notice also that the initialization occurs at the beginning of the program, and before any processing or memory allocation takes place. Finally, in order to eliminate any effects of operating system synchronization overheads, all of our applications use user-level spin locks.

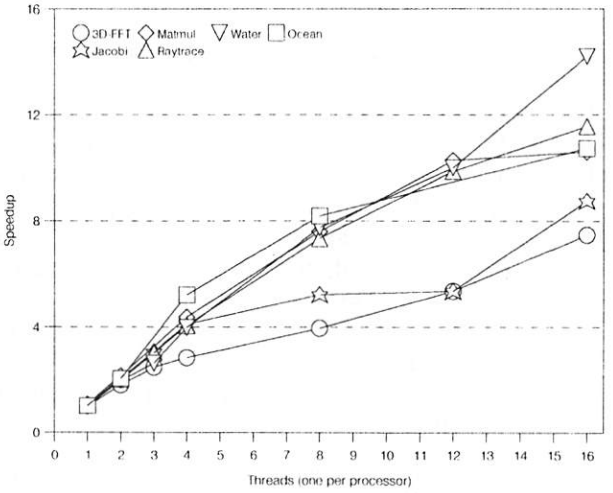


Figure 1: Best-case scalability

Figure 1 presents the best speedup that could be achieved as a function of the number of threads for each of the six applications. The base single-thread case was obtained by configuring the machine to run with a single processor and no remote memory. This base case corresponds to running the application on a uniprocessor machine containing the same amount of memory as an SMP node in our system, with the identical type of processor. The individual data points presented in this figure represent the best performance for each application and configuration over all thread and memory allocation policies. No one combination proved to be the best for all applications and thread configurations. Therefore, this figure serves to establish an upper bound on the scalability of our implementation.

Water exhibits the best scaling. This application has good locality in memory references, and not much sharing. Moreover, as the configuration grew larger, the aggregate total size of all L2 caches could store more of the application's working set. This effect also manifests itself in Jacobi, where the transition from 12 to 16 processors shows better scaling than from 8 to 12.

Three other applications (Matmul, Ocean and Raytrace) continued to improve as the number of processors increased, but the improvement started to taper off around 12 processors, due to a combination of bus bandwidth limitations and the effects of remote memory access. This phenomenon will be explained later in greater detail.

3D-FFT did not scale as well as the others. This application has substantial sharing among its threads causing it to scale poorly. Notice that 3D-FFT does not scale well on a single SMP.

Therefore it is reasonable to expect that it would not scale any better on larger machines.

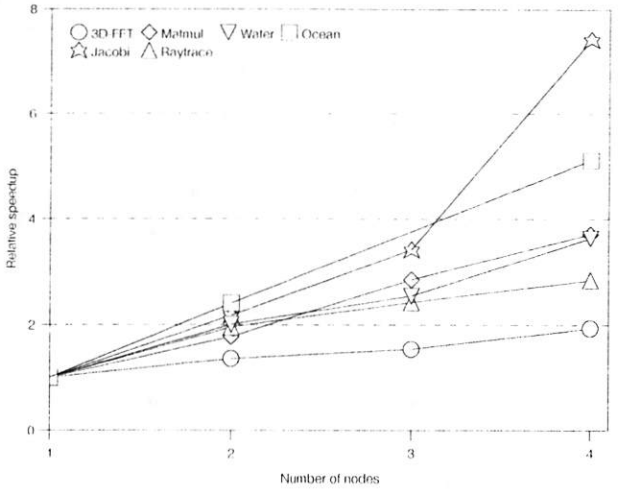


Figure 2: NUMA scalabiliy

Figure 2 shows the scalability of the applications on different ccNUMA configurations. This figure examines the benefits of implementing Intel-based multiprocessor machines larger than 4-processor SMP's. The baseline for this figure is the case where four threads execute on a single, standalone SMP node. The figure shows the scalability for the applications we studied as we run them on 8-way, 12-way, and 16-way ccNUMA machines. These are denoted in the figure by 2-node, 3-node, and 4-node, respectively. Each configuration was executed with as many threads as processors allocated **Round Robin**, and with memory allocated **Local**. Note that the scalability of some applications does not coincide with the best performance available. This is because the base case in Figure 2 is four threads running on a single SMP node, while the 4-thread data point in Figure 1 may correspond to an entirely different configuration. For instance, the best performing 4-thread case for Jacobi uses 4 nodes with the threads allocated **Round Robin** and **Local** memory allocation.

Figure 2 shows that NUMA scales approximately linearly for all applications except Jacobi. Ocean benefits from the increased collective cache capacity available in the system as nodes are added. Other applications such as 3D-FFT scale at a lower rate because of the increased inter-thread data sharing. Jacobi shows the best performance improvement as it scales from the 3-node to the 4-node configuration, because the aggregation of the L2 caches in the 4-node system can contain the entire working set of the application.

5.1 Effects of Allocation Policies

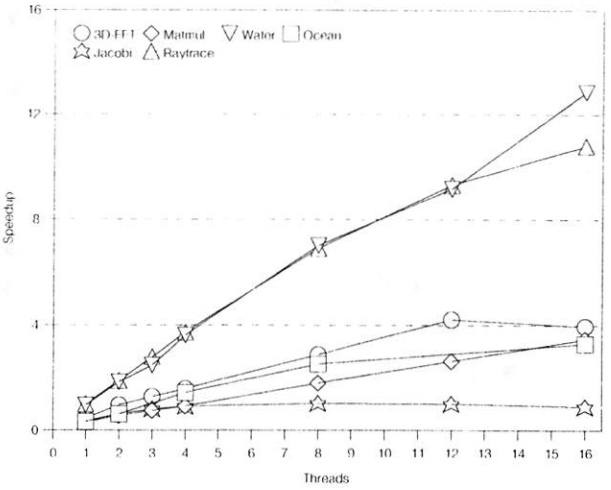


Figure 3: Base scalability: threads float, standard malloc

Figure 3 shows the scalability of the six applications when left unmodified. In this configuration, the threads are not bound to particular processors and memory allocation is done through the standard operating system mechanism. The results show that only two applications (Water and Raytrace) performed well in this configuration. The primary working sets of these applications fit within the L2 caches of the processors. Furthermore, there is no substantial sharing of data among threads during the computation. The remaining four applications did not scale well. For 3D-FFT and Jacobi, performance actually degraded when moving from 12-way to 16-way. This figure shows that scalability in performance will require application tuning and operating system support for memory affinity in ccNUMA machines built out of SHV components with no remote caches.

Figures 4 and 5 show the effect of tuning the applications to use Rsets. Two measurements are shown for each application, one with the application unchanged (broken line) and one with the application modified to benefit from Rsets (unbroken line). When the application is run unchanged, it uses NT's standard thread and memory allocation policies. When the application is tuned, it uses RSets to control thread and memory allocation. The thread and memory allocation policies used for these experiments are **Round Robin** and **Local,** respectively. The base single-thread case for these measurements was obtained by running the application with one thread on a single processor machine with no remote memory.

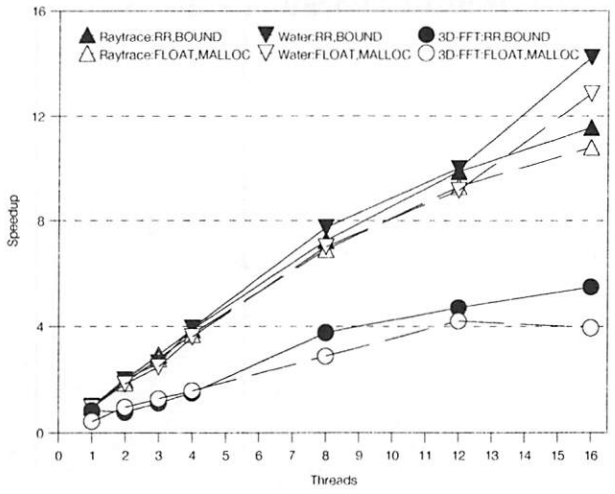


Figure 4: Applications exhibiting no affinity effects

Figure 4 shows that for 3D-FFT, Water, and Raytrace, there is not much to gain by modifying the application to use Rsets. Raytrace and Water work well with the caches in the system. Memory allocation in 3D-FFT is difficult to affinitize because over time, each thread accesses most of the application's data structures.

For these three applications, NT's default allocation policies yield good performance. Even though the unmodified versions of these applications do not specify a thread allocation policy, NT tries to reschedule each thread on the same processor on which it has recently run, so as to improve affinity in cache references [NT]. The NT memory allocator also worked well because we have observed that it uses page coloring to maximize the L1 and L2 cache performance. Since the primary working sets of Raytrace and Water fit within the cache especially in the large configurations, this policy yields good performance.

Figure 5 shows a different situation, where intelligent use of Rsets by the applications has a substantial impact on performance. When Rsets are not used, NT's default allocation policies do not allow the applications to overcome the effects of costly remote memory accesses. The difference in performance depends on the application, but in all cases, there is a clear gain from intelligently using Rsets. One can conclude from this figure that on ccNUMA machines that are architected out of SHV components with no remote caches, operating system support and tuning will be necessary to make these applications perform well. This result suggests that memory affinity support should become part of the standard Windows NT API if NT is to efficiently support this type of architecture. Although these conclusions depend on the fact that our system has no remote caches, it is not clear whether a remote cache will eliminate all the performance problems caused by NT's default memory allocation policies.

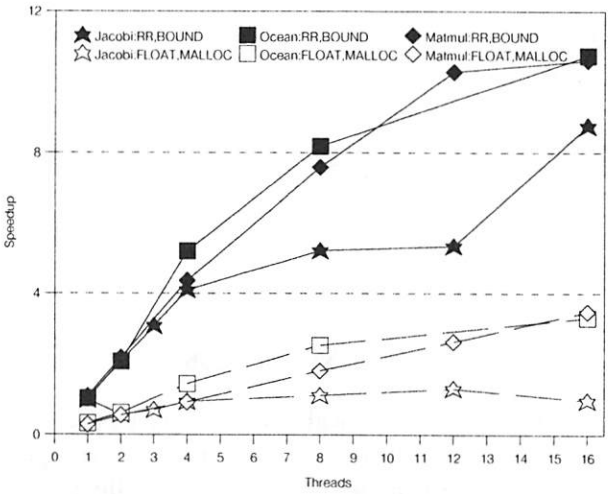


Figure 5: Applications exhibiting affinitization effects.

5.2 Effects of Local Bus Contention

Local bus contention is a serious problem as modern processors increase in speed. Figure 6 compares the speedup for the six applications in our suite under 3 different configurations. The configuration 1N4 was obtained by configuring the system as a 4-processor SMP, while 4N1 was obtained by configuring the system as a ccNUMA architecture with 4 nodes, each containing one processor. The 2N2 configuration was obtained by configuring the system as a ccNUMA architecture with 2 nodes, each containing two processors. Each configuration has exactly four processors. While 1N4 is a pure SMP system, 4N1 is a pure ccNUMA system. 2N2 represents a hybrid system.

The purpose of this experiment is to determine the impact that the local bus and the inter-node interconnect have on application performance. In the 1N4 case, the application threads face the maximum local bus contention and no interconnect effects. In the 4N1 case, the threads face the least local bus and maximum interconnect effects. The 2N2 case lies in between. Each thread was bound to a processor in this experiment. The memory allocation policy was **local**.

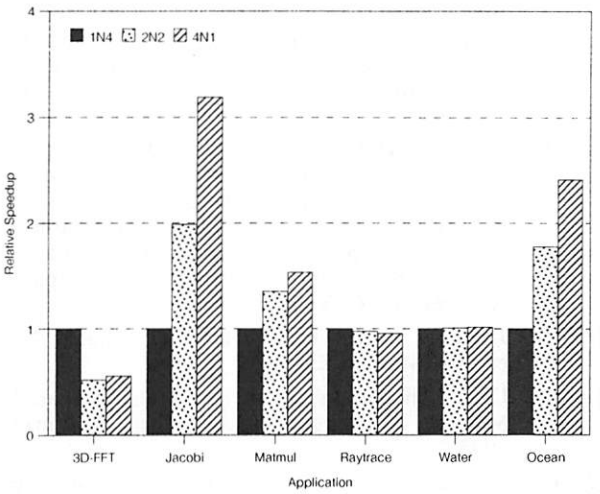


Figure 6: Local bus bandwidth effects.

For Raytrace and Water, there was not much of a performance difference across the different configurations. As mentioned earlier, these applications have small primary working sets that fit within the L2 caches, and exhibit modest inter-thread data sharing.

For 3DFFT, the effect of remote memory accesses dominates, and performance degrades as we move from an SMP to a ccNUMA architecture. The sharing pattern of this application is such that there is a continuous exchange of data among the threads. It is apparent that the bus bandwidth on a single SMP is adequate for the amount of data to be exchanged. Note that for this application, a remote cache would be of little help, because the contents of such a cache would be continuously invalidated as remote threads continue to modify the corresponding data.

For Jacobi, Matrix Multiply, and Ocean, the 2N2 and 4N1 configurations outperformed the SMP one. The reason is that the SMP case with four threads encounters significant local bus contention since the amount of data being accessed requires more bus bandwidth than is available with the existing Intel memory bus architecture. When the applications are modified to intelligently use RSets, remote memory accesses on the ccNUMA configurations have less of an effect. Therefore, the individual threads benefit from having less contention on the local bus.

5.3 Effects of Algorithmic Changes

Programs written to run in an SMP system will run without modifications on a ccNUMA system. However, it has been argued that NUMA-aware programs could further exploit the performance advantages of ccNUMA architectures, for instance by changing the algorithm used to exploit the characteristics of the NUMA environment (e.g. large amounts of physical memory). In our experiments, we have implemented a modified matrix multiplication algorithm in which the multiplier matrix is replicated at each node. Interestingly, this is similar to distributed algorithms that solve the same problem using message passing. Perhaps this suggests that transforming message-passing programs to run on NUMA systems will be more fruitful for performance tuning applications than just running existing SMP code.

6 Conclusion

We have built a 16-processor ccNUMA multiprocessor system using SHV 4-processor Intel Xeon SMPs. Windows NT was designed primarily to run on small SMP environments in which all processors have equally fast access to all the system memory. It therefore faces performance challenges in an environment where the processor-to-memory speed varies across a system. To overcome these problems and enable NT to run in this environment efficiently, we implemented an abstraction called the Resource Set that allows threads to specify where memory is to be allocated across a NUMA complex. Thus, threads can specify that memory should be allocated from banks that are close to the processors on which they run. This affinity in memory allocation can result in several performance benefits when running parallel applications. Our results indicate:

- The approach of building ccNUMA architectures out of SHV components is viable. For five out of six applications we studied, performance continued to improve in various degrees as we increased the number of processors. In general, it seems that in many cases the penalty of remote memory accesses can either be masked or does not have much of an effect to begin with.
- On architectures such as ours, where there are no special hardware assists or remote caches, scaling the performance of the application may require application tuning and operating system support for memory affinity.
- Memory allocation affinity should become a part of the standard API of Windows NT, as ccNUMA machines become increasingly common.
- For some applications, local bus saturation is the dominant performance impediment. This is somewhat surprising since one would expect

the latency of the NUMA interconnect to be a significant source of overhead in a ccNUMA system.

- Generally speaking, applications that scale well on an SMP system seem to also scale well on a ccNUMA environment, and vice versa.

Acknowledgments

We would like to thank HAL Computer Systems and Fujitsu for providing us with the MCU that we have used in our prototype, and also for providing their 2-node eBIOS and HAL code, which we extended to support a 4-node system. We also would like to thank the anonymous referees for their comments and Rich Oehler from the program committee for his help.

Appendix

The tables at the end of this paper contain the raw data used to create Figures 1 through 6.

Table 1 lists the execution times of the applications for different 16-processor configurations. All times are in seconds. BEST refers to the best-case scalability. For each data point, the configuration that yielded the least execution time is also provided. The BASE case provides the execution times when standard NT allocation policies are used. The TUNED case presents the execution time when the applications intelligently make use of Rsets. The data in this table was used to create Figures 1, 3, 4, and 5. The base single-thread execution time used to generate speedup numbers is given under each application. The single-thread execution was measured on a 1N1,t3,m0 system.

Tables 2 lists the execution times of the applications for different NUMA configurations. All times are in seconds. The allocation policy in each case is t0,m2. The data in this table was used to create Figures 2 and 6.

References:

[Alewife] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B. H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife Machine: Architecture and Performance. Proceedings of the 22nd International Symposium on Computer Architecture, June 1995.

[Brazos] E. Speight and J. K. Bennett. Brazos: A third generation DSM System. Proceedings of the 1997 USENIX Windows/NT Workshop, August, 1997.

[DG] Data General's NUMALiiNE Technology: The Foundation for the AV 25000 Server, http://www.dg.com/about/html/av25000_foundation.htm

[FJST] W. Weber, S. Gold, P. Helland, T. Shimizu, T. Wicki, and W. Wilcke, The Synfinity Interconnect Architecture: A Cost-Effective Infrastructure for High-Performance Servers, Proceedings of ISCA'97, the 24th Proceedings of the Annual International Symposium on Computer Architecture.

[Flash] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH Multiprocessor. In Proceedings of the 21st International Symposium on Computer Architecture, Chicago, IL, April 1994.

[Gray] Jim Gray, ed., The Benchmark Handbook, 2nd edition, Morgan Kaufman, 1993.

[Len] D. E. Lenoski and W. Weber, Scalable Shared-Memory Multiprocessing, Morgan Kaufman, 1995.

[MSCS] Vogels, W., Dumitriu, D., Birman, K. Gamache, R., Short, R., Vert, J., Massa, M., Barrera, J., and Gray, J., "The Design and Architecture of the Microsoft Cluster Service - A Practical Approach to High-Availability and Scalability", Proceedings of the 28th symposium on Fault-Tolerant Computing, Munich, Germany, June 1998.

[NT] David Solomon, Inside Windows NT, 2nd edition, Microsoft Press, 1998.

[Sch] Martin Schulz and Hermann Hellwagner, Extending NT Virtual Memory by SCI-based Hardware DSM 2nd USENIX Windows NT Symposium, August 1998, Seattle WA, USA.

[SGI] James Laudon and Daniel Lenoski, The SGI Origin: A ccNUMA Highly Scalable Server, Proceedings of the 24th International Symposium on Computer Architecture, 1997, pp 241-251.

[Shrimp] Yuanyuan Zhou, Liviu Iftode and Kai Li, Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems, Second Symposium on Operating System Design and Implementation, Seattle, WA, 1996.

[Spl] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. In Computer Architecture News, vol. 20, no. 1, pp 5-44.

[Spl2] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In Proceedings of the 22nd International Symposium on Computer Architecture, Santa Margherita Ligure, Italy, June 1995, pp 24-36.

[Sting] Tom Lovett and Russell Clapp, STiNG: A CC-NUMA computer system for the commercial marketplace, Proceedings of the 23rd Annual International Symposium on Computer Architecture, May 1996

[TPC] Transaction Processing Performance Council, http://www.tpc.org.

[Uni] Unisys, Cellular MultiProcessing Architecture, http://www.marketplace.unisys.com/ent/Cmpwh.pdf.

[Vas] Raj Vaswani and John Zahorjan. The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared Memory Multiprocessors. Proc. 13th Symposium on Operating System Principles, October, 1991.

[Ver] Ben Verghese, Scott Devine Anoop Gupta, and Mendel Rosenblum, Operating system support for improving data locality on CC-NUMA compute servers. Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems, October 1996.

Trademarks

App	Config	1-thread	2-thread	3-thread	4-thread	8-thread	12-thread	16-thread
3D-FFT 39.60	BEST	39.00 1N4,t3,m0	21.90 1N2,t3,m0	15.80 1N3,t3,m0	13.90 1N4,t3,m0	10.00 4N2,t0,m5	7.40 4N4,t1,m5	5.30 4N4,t0,m5
	BASE-NT	89.50	41.00	31.00	25.10	13.70	9.40	10.00
	TUNED	46.80	50.30	35.40	26.40	10.50	8.40	7.20
Jacobi 86.70	BEST	83.90 4N4,t0,m2	41.80 4N4,t1,m2	28.20 4N4,t1,m2	21.10 4N4,t1,m2	16.60 4N4,t1,m2	16.20 4N4,t1,m2	9.90 4N4,t0,m2
	BASE-NT	252.30	142.40	107.30	96.80	84.30	87.50	97.70
	TUNED	83.90	41.80	28.20	21.10	16.60	16.20	9.90
Matmult 102.50	BEST	97.10 1N4,t3,m1	56.00 1N2,t0,m1	38.90 1N3,t3,m1	33.50 1N4,t3,m0	42.40 4N4,t0,m2	32.40 3N4,t0,m2	28.10 4N4,t0,m2
	BASE-NT	358.20	186.30	138.50	112.20	56.80	38.90	29.50
	TUNED	102.20	143.40	114.90	93.60	49.50	34.90	28.10
Raytrace 114.50	BEST	110.70 4N4,t3,m1	56.60 1N2,t0,m1	37.70 1N3,t0,m1	28.20 1N4,t0,m1	15.60 4N4,t0,m5	11.60 3N4,t0,m2	9.90 4N4,t0,m1
	BASE-NT	116.10	60.60	40.90	30.70	16.60	12.30	10.60
	TUNED	116.80	57.80	39.00	29.40	15.80	11.60	9.90
Water 239.10	BEST	237.70 1N4,t3,m1	119.60 2N1,t0,m2	90.10 3N1,t0,m2	60.10 4N1,t0,m2	30.90 4N4,t1,m2	23.90 4N3,t0,m2	16.80 4N4,t0,m2
	BASE-NT	252.80	130.10	96.30	66.00	34.10	26.10	18.60
	TUNED	238.50	119.90	90.50	60.60	30.90	23.90	16.80
Ocean 17.20	BEST	16.80 4N4,t0,m2	8.30 4N4,t1,m2	x	3.30 4N4,t1,m2	2.10 4N4,t1,m2	x	1.60 4N4,t0,m2
	BASE-NT	52.60	27.90	x	12.00	6.80	x	5.20
	TUNED	16.80	8.30	x	3.30	2.10	x	1.60

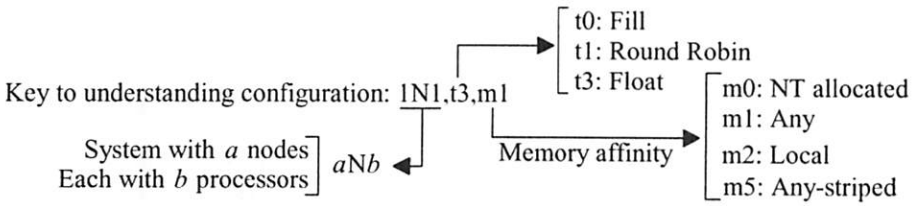


Table 1: Execution times for base, best, and modified cases.

Application	4N1	2N2	1N4	2N4	3N4	4N4
3D-FFT	24.90	26.70	14.00	10.30	9.10	7.20
Jacobi	23.00	36.90	73.40	33.80	21.50	9.90
Matmult	23.48	26.70	36.25	20.40	12.66	9.75
Raytrace	29.30	28.90	28.20	14.40	11.60	9.90
Water	60.10	60.20	61.20	30.40	24.00	16.80
Ocean	3.40	4.60	8.20	3.40	x	1.60

Table 2: Execution times for different t0,m2 NUMA configurations

The Record-Breaking Terabyte Sort on a Compaq Cluster

Samuel A. Fineberg
Sam.Fineberg@Compaq.com
Pankaj Mehra
Pankaj.Mehra@Compaq.com
Compaq Tandem Labs
19333 Vallco Parkway, M/S CAC01-27
Cupertino, CA 95014 USA

Abstract

Sandia National Laboratories (U.S. Department of Energy) and Compaq Computer Corporation built a 72-node Windows NT cluster, which Sandia utilizes for production work contracted by the U.S. government. Recently, Sandia and Compaq's Tandem Division collaborated on a project to run a 1-terabyte commercial-quality scalable sort on this cluster. The audited result was a new world record of 46.9 minutes, three times faster than the previous record held by a 32-processor shared-memory UNIX system. The external sort utilizes a unique, scalable algorithm that allows near-linear cluster scalability. The sort application exploits several key hardware and software technologies; these include dense-racking Pentium II based servers, Windows NT Workstation 4.0, the Virtual Interface Architecture, and the ServerNet I System Area Network (SAN). The sort code was highly CPU-efficient and stressed asynchronous and sequential I/O and IPC performance. The I/O performance when combined with a high-performance SAN, yielded supercomputer-class performance.

1. Introduction

High-performance sorting is important in many commercial applications that require fast search and analysis of large amounts of information (for example, data warehouse and Web searching solutions). We present the design of a cluster and a commercial-quality sorting algorithm that together achieved record-breaking performance for externally sorting a terabyte of data. The audited terabyte-sorting time of 46.9 minutes was three times faster than the previous record [NyK97], which was achieved on a 32-processor SGI Origin 2000.

The cluster consists of 72 Compaq Proliant™ servers running Microsoft Windows NT™, although only 68 of the servers were utilized for the sort. It was built in collaboration with US Department of Energy's Sandia National Laboratories in New Mexico, where it is currently deployed for production work contracted by the U.S. government. The cluster nodes communicate using Compaq's ServerNet-I SAN (System Area Network). The SAN topology uses unique dual-asymmetric network fabric technology for performance and reliability [MeH99].

The external three-pass parallel sorting algorithm is also expected to scale well on larger clusters. In fact, the algorithm was originally developed for Compaq's NonStop™ Kernel (NSK) based servers (and will be productized on both NSK and Windows NT clusters). The program uses an application-specific portability library, libPsrt, and a message-passing library, libVI. Section 4 describes the application software architecture in detail. LibVI builds upon the VI Primitives Library specified by the Virtual Interface Architecture (VIA) Specification [Com97]; it provides a robust, high-throughput, multi-threaded and thread-safe messaging layer with efficient waiting primitives based on NT's IO Completion Ports. LibPsrt builds upon libVI and supports more application-specific operations, such as remote I/O and memory management.

2. Cluster Hardware Configuration

The cluster hardware consisted of rack mounted Proliant servers, a ServerNet™ I SAN, and over 500 disks. The disks were either internal to the nodes or plugged in to Compaq hot-pluggable drive enclosures. The total purchase price for a similarly configured 68-node cluster would be $1.28M.

This system consists of the following hardware:

Item	Quantity
Servers	68 systems, 136 CPUs, 34GB RAM
Disks	269 Wide Ultra SCSI-3 busses, 537 disks, 62 external disk enclosures, 4.5TB storage
ServerNet	68 dual-port ServerNet I NICs, dual-fabric topology utilizing 48 6-port ServerNet I switches
Racking	22 racks, 12 KVM switches, 2 rack mounted flat-panel monitors
Operating System	67 Windows NT Workstation, one 70-license Windows NT Server
Ethernet Network	68 Embedded 100BaseT NICs (included in the servers) attached to a single Cisco Catalyst 5500 switch

The costs break down as shown in Figure 1. As shown in the graph, the largest portion of our system's cost was in disks and external disk enclosures. The second largest portion was the servers (including additional CPUs and memory added onto the base server configuration). The ServerNet network was only 16% of the system cost, including all of the cables, switches, and NICs. The racking gear, KVM (keyboard, video, mouse) switches, and the monitors used as system consoles contributed only 5% to the cost. The 72-port Cisco Catalyst 5500 Ethernet switch made up 3% of the system cost, and finally, the Windows NT operating system made up 2% of the system cost.

2.1 Compaq Proliant 1850R Servers

The full Sandia cluster contained 72 rack-mounted Compaq Proliant 1850R servers, each with:

- Two 400MHz Intel Pentium II Processors
- 512MB, 100Mhz SDRAM
- One integrated and one PCI-card based dual-Ultra-Wide SCSI-3 controller (for a total of 4 SCSI busses)
- One dual-ported ServerNet I NIC

Of these nodes, 68 were actually used for the sort: 67 as sort nodes and one as the sort manager. In addition, a total of over 500 SCSI disks (in varying configurations) were distributed among the 67 sort nodes (nominally, 8 disks per sort node — 7 for sorting and one for the OS). The disks were all striped using Windows NT's ftdisk utility. Each node ran Windows NT Workstation 4.0 Service Pack 3 (except the sort manager node which ran

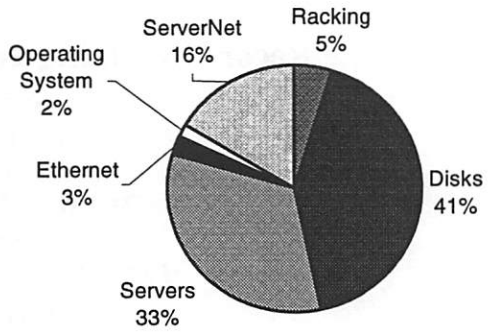


Figure 1: Cluster cost breakdown

NT Server). All nodes were running a ServerNet-specific software stack comprising Compaq ServerNet I VI (SnVie) version 1.1.10, ServerNet PCI Adapter Driver (SPAD) version 1.4.3, and ServerNet SAN Manager (SANMAN) version 1.1.

The Proliant 1850R nodes were ideal for this cluster because of several key features: high memory bandwidth, small rack footprint, and integrated remote console for system management. Memory bandwidth of 450MB/sec was measured with the STREAMS benchmark on these nodes (using both CPUs). This is vital to achieving good sorting performance because much of the time spent sorting is spent in merges, which tend to stress processor-memory bandwidth in a manner similar to the long-vector accesses modeled by the STREAMS benchmark. The small footprint was important in order to make such a large cluster feasible. Proliant 1850Rs take only 3U (about 5.25 in/13.3 cm) of rack space, and include 3 1-inch hot-pluggable drive bays, room for 2 internal drives, and 4 PCI slots. Finally, the integrated manageability features allowed us to monitor and reboot nodes from a single system console using Compaq Insight Manager software.

2.2 ServerNet I SAN

Each ServerNet I PCI NIC provides two ServerNet ports: an "X" port and a "Y" port. Each port contains a transmitter and a receiver, both of which can operate concurrently, driving a bi-directional parallel LVDS cable in both directions simultaneously. The NICs in our system are interconnected by two fabrics of 6-port ServerNet-I crossbar routers (see Figure 2). An "X fabric" connects all the X ports, and a "Y fabric" connects all the Y ports. The two router fabrics are complete but asymmetric: each node interfaces with both fabrics and each fabric interfaces with every node, but the topologies of the two fabrics are different. Therefore, each fabric provides a path between every source-destination pair, but the path lengths (measured

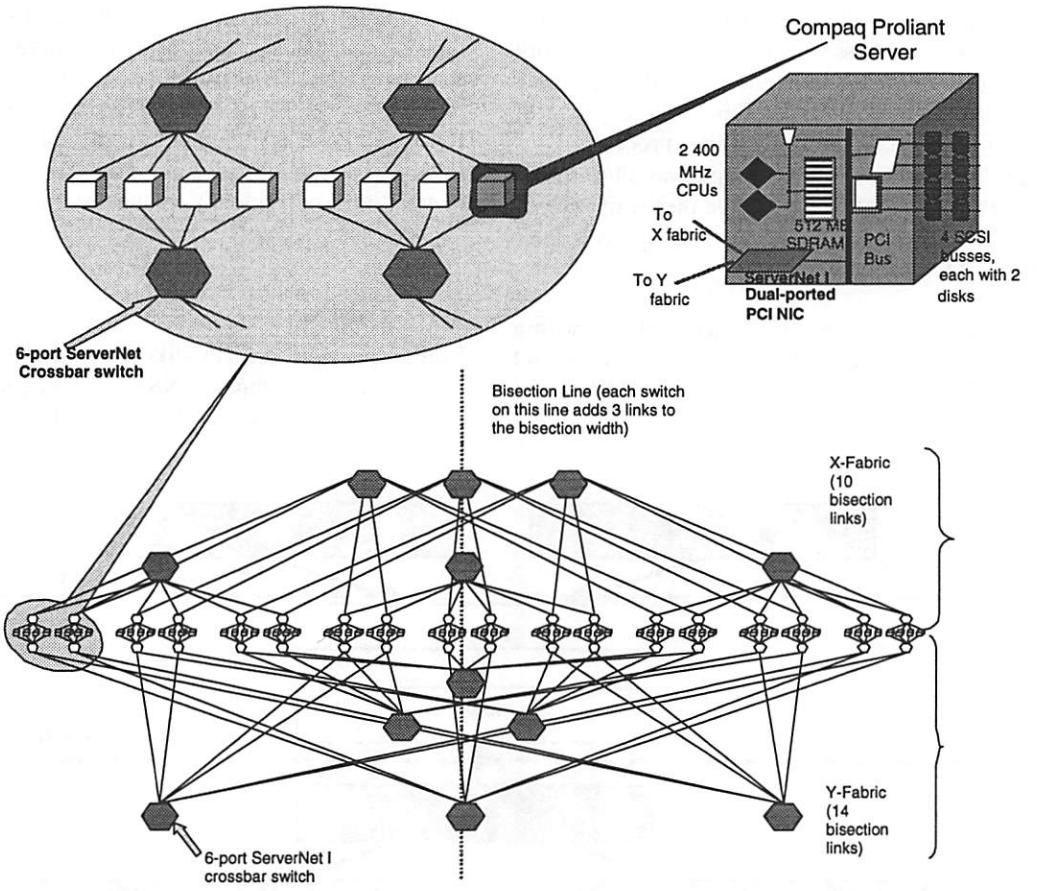


Figure 2: Sandia network topology

in router hops) of most paths differ from one fabric to the other.

Traditionally, ServerNet topologies used two identical fabrics for reliability (but not for performance) by switching to a path in the Y fabric when the X fabric path to a destination was down, and *vice versa*. Here, we utilize the two fabrics for reliability *and* performance by configuring SnVie to "prefer" the shorter path when establishing a connection between two processes on different end nodes. This reduces network-routing latency (300 ns pipelined latency per packet per ServerNet I router) by creating and using paths with fewer hops; we also lower the probability of output-port contention (because each fabric handles only half of the total message traffic). Fault tolerance is maintained by this approach because there are at least two possible paths between each node pair, and traffic can always be routed via the alternate path if the preferred path fails.

3. The Sort Algorithm

David Cossock of Tandem Labs developed sort, merge, and parallel selection algorithms [Cos98] that were used. The file to be sorted is distributed across the local filesystems of the sort nodes, and sort processes are assigned to nodes. While the algorithm applies equally well to variable-length records, the results reported here are with files containing fixed-width records (80 bytes long) each containing an 8-byte (uniformly distributed) random integer key. The sort algorithm leaves the sorted data distributed among nodes such that the node-order concatenation of all the files consists of records sorted by the integer key in either ascending or descending order.

The three sort phases include the following:

- Phase 1 (local-sort): each node sorts memory-resident runs of the local partition.
- Phase 2 (local-merge): the local runs are merged so that each node has a completely sorted work file.

- Phase 3a (partition): The sort processes execute a parallel selection protocol over ServerNet, determining (1) the upper and lower bounds of their ultimate sorted output partition, and (2) the relative byte addresses, within each node's work file, of the beginning and end of records containing keys between these bounds. This is the only part of the sort that utilizes the sort monitor process.
- Phase 3b (parallel-merge): Using remote I/O across the ServerNet SAN, nodes merge pieces of each of the other nodes' work files to end up with their portion of the sorted data.

While Phases 1 and 2 stress the system balance within each node, stressing its CPU-memory and disk I/O subsystems, Phase 3b stresses each node's I/O bus and the cluster interconnect.

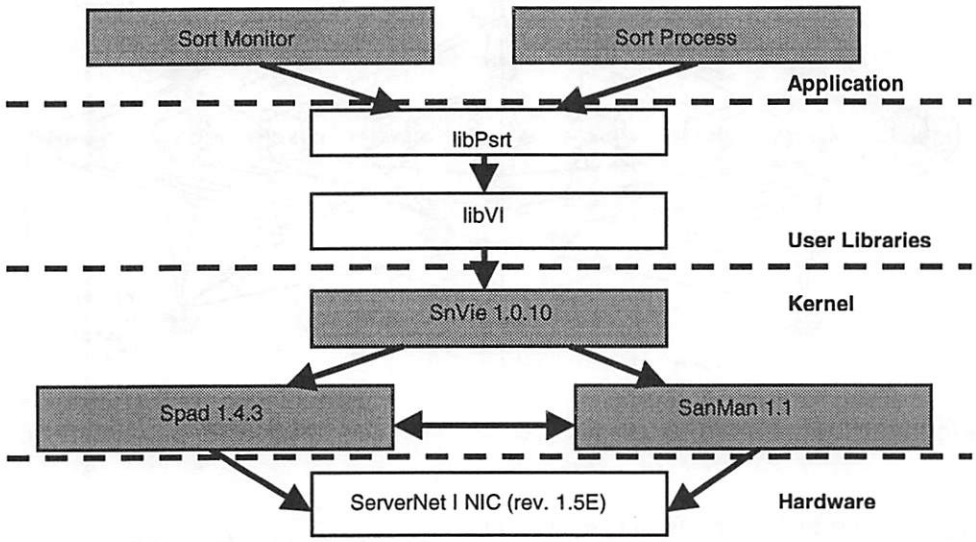


Figure 3: Sort Communication Architecture

4. Communication Software Architecture

The architecture of the sort application is shown in Figure 3. Each sort node runs a "sort process", and the sort manager node runs the "sort monitor" process. Each sort process contains two threads. A "sorting thread" implements the sorting algorithm minus remote I/O and interprocess communication (IPC), and a "shadow thread" performs, in the background, IPC and remote I/O (on behalf of other sort processes). Each of these threads uses a library called libPsrt, which provides mechanisms for both IPC and remote I/O. Local I/O is handled directly by the sort thread itself using NT's asynchronous ReadFile and WriteFile system calls and event-based completion. All remote I/O and IPC is also asynchronous and uses Windows NT I/O Completion Ports (IOCPs) to determine operation completion. This means that each sorting thread can issue commands, which can be IPC, remote I/O, or inter-thread (within a process) communication, and then it can wait on its own I/O completion port for the next completed operation or incoming request. The shadow thread is essentially a dispatch loop that receives requests and completion notifications on one IOCP and posts completion of local requests on another IOCP to be read by the local sorting thread. It also posts completions of remotely requested operations on VI connections (as "reply" messages).

4.1 LibPsrt

LibPsrt provides an RPC-like model, based on the I/O architecture of Compaq's NSK operating system (note: we could not utilize Windows NT RPC because it did not support the ServerNet SAN). In this model, sort and shadow threads issue "requests" to other threads that can be the same node or on any other node in the cluster. Each of the requests contains a command and a data portion. When the request is fulfilled, the thread sends a "reply", which also includes a data portion. An example of this is when the sort manager wants to obtain the "key" ranges in order to do partitioning in Phase 3a of sorting. It sends a message to each of the nodes with a "key request" command and a blank data segment. Each node replies to key request with its keys included in the reply's data segment. Because the sort process is not always ready to process requests, all IPC requests are first handled by the shadow thread, which forwards them to the sorting thread (i.e., it posts them to the sort thread's IOCP). Then, when it is ready, the sort thread reads these requests from its IOCP, and

replies directly with the requested data. For remote I/O requests, the shadow thread reads the local sort file and responds directly to the requesting remote thread without involving the sort thread.

4.2 LibVI

LibPsrt is built on top of libVI, a message passing library built on the Virtual Interface Architecture (VIA). LibVI provides a traditional send/receive style messaging API. It is thread safe, and has support for asynchronous send/receive operations (each implemented as a synchronous call in its own thread). One key feature of libVI is its ability to post message completions through Windows NT I/O completion ports. For a more detailed description of libVI, see [Fin99].

4.3 ServerNet Drivers

LibVI uses SnVie, a kernel-level implementation of the Virtual Interface Architecture for ServerNet I hardware [Hor95]. It is intended as a porting vehicle for migration to ServerNet II [HeG98][1]. While SnVie provides all of the features of ServerNet II, it is implemented in software. Therefore, it has lower performance than ServerNet II VI, which will be implemented in hardware. In ServerNet II, VI data transfers will occur without any kernel transitions. LibVI will run without modification on ServerNet II when it becomes available. SnVie uses ServerNet I's TNet Services API, which is supported by the ServerNet PCI Adapter driver (SPAD). Name service support for low-level ServerNet Node IDs is provided by the SAN Manager driver (SANMAN).

5. Performance Studies with the Sandia Cluster

5.1 Sequential I/O Performance with Striped Disk Partitions

Windows NT offers several mechanisms for issuing and completing I/O requests. In this section we present some performance measurements that were made during sort application development. These results represent filesystem and disk performance under Windows NT, but not the performance of the sort application code. However, we utilized these measurements to make better choices when implementing the sort.

[1]for more information on ServerNet II see http://www.servernet.com

Riedel. et al [RiV98] report that striping large accesses across multiple disks, using unbuffered I/O, and having many outstanding requests are the ways to optimize sequential disk performance. Benchmark code for their work is also available from Microsoft Research (http://research.microsoft.com/barc/Sequential_IO/). We augmented their benchmark code with support for IOCPs (I/O Completion Ports), an efficient mechanism for notification of outstanding I/O requests. Riedel, *et al* had found write performance saturating below read performance.

Using this benchmark, we found that:

- The best performance was given by asynchronous unbuffered I/O using either event-based or IOCP signaling;
- At 512KB I/O request size, peak read performance and very nearly the peak write performance could be achieved with just 2 I/O requests outstanding at any given time; otherwise, 8 outstanding I/O requests were needed for full throughput at 128KB I/O size; and
- We needed to stripe I/Os across either 2 10K-rpm or 3 7200-rpm disk drives in order to fully exploit a 40 MB/s Ultra-Wide SCSI bus.

In fact, a single Seagate ST39102LW (9 GB, 10K-rpm) drive was measured at 18.43 MB/s at 128KB I/O size with 8 outstanding requests under an Adaptec SCSI controller.

The Compaq Proliant 1850R nodes have an integrated Symbios 53c875 dual-channel SCSI controller. With 3 10K-rpm drives striped across two SCSI strings, at 512KB I/O size, and 20-deep asynchronous unbuffered I/O issue, we were able to reach a total bandwidth — reading or writing — of 53 MB/s. The deployed system has a second dual-channel SCSI controller in the form of Symbios 53c876 PCI card. At about 110MB/s the 32-bit 33-MHz PCI bus peaks; the four SCSI strings with 3 disks each are therefore quite enough to exploit all available I/O bus bandwidth in the server.

5.2 Communication Performance with LibVI over ServerNet I VI

5.2.1 ServerNet Hardware Performance and Scalability

Each ServerNet I link is rated at 50 MB/s in each direction. ServerNet I hardware-level packets contain up to 64 bytes of payload and 16 bytes of header and CRC. Each packet specifies either a read/write request

or a read/write response. Only read responses and write requests carry payload. Requests and responses occur in one-to-one correspondence; *i.e.*, every request must be *acknowledged* by a response. It is easy to see that peak *uni*-directional data bandwidth on a ServerNet I link is at most 40 MB/s when we consider packetization and acknowledgement overheads; likewise, peak *bi*-directional data bandwidth on a ServerNet I link is 33.3 MB/s.

Each node is allowed to have up to 8 request packets outstanding. The SPAD driver configures this limit to 4. The ServerNet I NIC responds to requests received from the network by performing PCI read or write operations. It is important to note that, irrespective of the request type, every request-response sequence involves exactly one PCI (memory space) read transaction and one write transaction. While fewer than 4 requests are outstanding, an end node can continue to generate requests. Once the limit is reached, an end node can only issue responses; the next request must wait for a response.

Data transfer occurs when an initiating node uses its Block Transfer Engine (BTE) to initiate a request; at the target node, the Access Validation and Translation (AVT) logic carries out the specified operation and generates a response. ServerNet routers perform wormhole routing, allowing packet transfer to occur in a pipelined fashion: Each packet travels through the network as a train of symbols; the routing logic is pipelined so that within 300 nanoseconds of arriving at an input FIFO on the switch, the head of the train emerges at an output port with the remaining symbols following one per 50 MHz clock. So long as four or more requests can be completed per round-trip time, the pipeline will continue to run efficiently.

There are three principal sources of "pipeline bubbles" in ServerNet I: (1) PCI bus first-byte read latency, (2) single-threadedness of the BTE engine, and (3) output-port contention in routers.

The first of these reduces the average PCI efficiency of ServerNet I NICs, pushing the 32-bit 33-MHz PCI bus into saturation as soon as a node hits 33 MB/s uni-directional outbound traffic or 19 MB/s bi-directional traffic. PCI saturation can delay generation of response packets, causing packet round-trip times to shoot up.

The second causes the BTE to idle when all 4 requests are outstanding, pushing the network interface into an outstanding request (OR) limited mode, where the peak bandwidth of a connection drops to 4*64/RTT MB/s where RTT is the hardware round-trip time for a request-response pair in microseconds. Another property of a single-threaded BTE is its inability to use both fabrics simultaneously; at best, it can support static load balancing of connections between fabrics.

Output-port contention occurs in ServerNet routers when two packets try to go out of the same router port; one of them is blocked, causing other traffic behind it to back up. Thus, output-port contention causes congestion in the network, lowering the network's link utilization and throughput and increasing the RTT of packets. We hasten to note that ServerNet II has much better performance characteristics on all three counts.

5.2.2 LibVI Performance and Scalability

In this section we present raw performance measurements of the libVI communication library running over ServerNet I VI. We measured both point-to-point and all-to-all performance, though the sort primarily stresses the all-to-all bandwidth or the network.

5.2.2.1 Point-to-point latency and bandwidth

To measure point-to-point performance a simple ping-pong test was utilized. This test consisted of one process sending a message of a given size to another process, then that process sending the message back. The time for this operation was halved to get the one-way send time. For this test, all data were sent synchronously using blocking send and receive functions. In addition, all graphs in this paper use the VIA send/receive-based long message protocol (rather than the VIA remote-DMA based protocol) as described in [Fin99] because it performed better for sort phase 3b.

LibVI's basic message latency was about 138 microseconds. Note that ServerNet I VI is a software implementation, and this latency is system dependent.

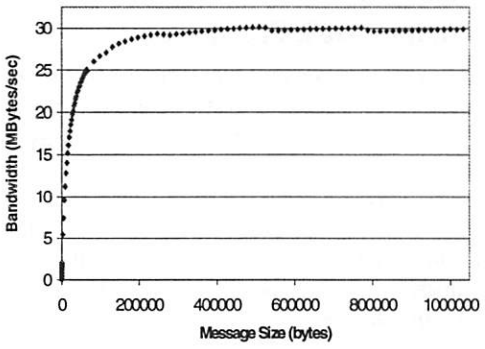


Figure 4: Point-to-point libVI bandwidth

Latencies between 80 and 200 microseconds have been observed with libVI on different systems. While this variation should decrease with ServerNet II's hardware VIA implementation, PCI and memory latency as well as CPU speed will always cause this to vary.

One of the primary advantages of VIA based networks is their ability to transfer long messages using DMA, freeing the processor to perform computation or to service short messages. LibVI was designed to minimize CPU utilization, and maximize the use of DMA. This was done primarily by using "wait" commands to check for message completion, rather than polling for completion. This implementation results in extremely low CPU utilization. For long messages, libVI's CPU utilization was less than 5% running at full bandwidth. Even for short messages, CPU utilization was quite low. However, this was done at the cost of message latency. It is likely that lower latency could be achieved with a polling-based short-message protocol, or a protocol that does not have an additional thread involved in message receipt. However, our protocol seeks to maximize concurrency and message throughput, a feature well suited to data-intensive applications. In fact, this sort was quite insensitive to latency. Sort performance was almost entirely a function of SAN and disk bandwidth. With such a low CPU utilization, it is possible to effectively overlap operations in the CPU with communication to an extent not possible with most message-passing libraries and networks.

Message bandwidth is shown in Figure 4. The point-to-point bandwidth achieved was 30MB/s. This difference between this and the "achievable peak" of 33.3MB/sec was due to the "long-message" protocol used for this measurement.

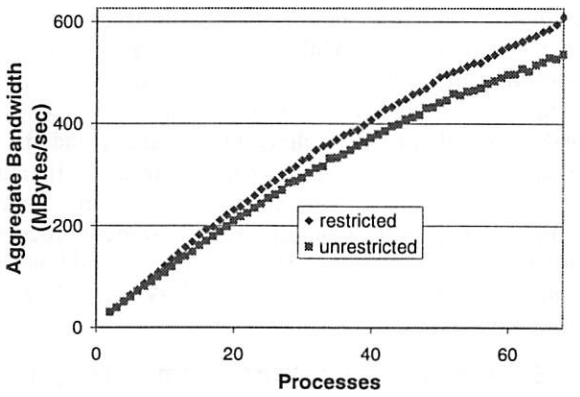


Figure 5: Long message aggregate bandwidth protocol comparison

LibVI internally can utilize either VIA's send/receive mechanism or its remote DMA mechanism for transferring messages larger than 56 bytes (small messages are always transferred with send/receive). In our experiments, VIA send/receive delivered better aggregate bandwidth when many nodes were communicating with many nodes simultaneously. However, point-to-point bandwidth of the send/receive based protocol was about 10% worse than the remote DMA based protocol (with only one pair of nodes communicating across the entire cluster). Because the sort performance was more dependant on aggregate bandwidth, we chose to use libVI's send/receive based long-message protocol for the sort. This protocol is a compile-time internal option for libVI, and does not change libVI's semantics or its API.

5.2.2.2 Aggregate bandwidth

One of the key metrics against which libVI was benchmarked is aggregate bandwidth. This was especially important for the Terabyte Sort code, which included a parallel-merge phase limited by the network's aggregate bandwidth. While these results were limited by the networking hardware utilized in the Sandia cluster, we also found some non-obvious protocol effects.

The aggregate bandwidth test utilizes the IOCP send/receive routines. The test initially issues one 512K byte send to every other process and one 512K byte receive from every process (the 512KB message size was chosen because it was the block size used in the parallel-merge phase of the sort). Then, it sits in a loop waiting on the IOCP for one of the operations to complete. As operations complete, the program re-issues the same operation (either or an asynchronous send or receive) to or from the same process. It continues to re-issue the operations until each specific send and receive has been issued 50 times. This results in the transmission of 25MB*<# of procs> of data both in and out of every process in the cluster. Because messages are issued in completion order, the communication pattern is random and there is no attempt made to alleviate hot-spots. This unstructured communication is very similar to the sort's parallel merge, and was a good predictor of sorting performance.

We ran this test with one process per node across the set of nodes from 0 to N-1 (where N is the total number of processes on which we ran the test). This resulted in the performance shown by the squares in Figure 5 (i.e., the "unrestricted" line). As can be seen from the graph, scaling is relatively linear, but per-node bandwidth is below the peak ServerNet bandwidth of 33MB/sec per direction.

In fact, we were achieving only about 8 MB/sec/node (bi-directional) vs. the "achievable" maximum, 19 MB/sec (bi-directional). The primary cause of this was that the network was not a fully connected crossbar. The topology we utilized was only capable of about ½ of the peak bandwidth, so we should only be able to attain about 9.5MB/sec/node.

However, 8MB/sec/node was still slower than what we expected. This turned out to be from node contention. Essentially, if two or more nodes are sending or receiving data from a node in the system, then they will receive it slower than if they had sequentialized their access to that node. This is particularly true for the sort, where every node is likely to have multiple send/receive operations that are ready to begin at any given time. While it is combinatorially unlikely that 3 or more nodes will be reading from or writing to any node's ServerNet NIC, it is quite likely that there will always be some NIC that has at least two nodes contending for it. Unfortunately, it was impossible to orchestrate the communication in the sort code to fully alleviate this problem, but it was possible to impose restrictions on the libVI long-message protocol.

The changes to the libVI communication protocol introduced "restrictions" in different portions of the protocol. First, consider the send/receive protocol. In the standard "unrestricted" protocol, it is possible to be simultaneously receiving a message from and sending a message to every other node in the cluster. However, we can impose a lock that restricts this to a single "long-message" send and a single "long-message" receive active at any given time. The results of these changes are shown in the diamonds in Figure 5. As can be seen, for small numbers of nodes, the unrestricted protocols performed slightly better, but for larger number of nodes, the restricted send/receive performed significantly better. This difference was an increase in the bi-directional bandwidth of about 1MB/sec/node for the full machine, raising performance to about 9MB/sec/node. Therefore, the restricted send/receive protocol was used for the Terabyte Sort. However, for typical applications that do not perform the same amount of unstructured all-to-all communication, the unrestricted protocol is recommended since it performs better in the more common case.

5.3 Sort Application Performance

The following sorting performance was obtained on 68 nodes of our 72-node cluster. One node was dedicated to a sort monitoring process. Each of the remaining 67 nodes owned a partition consisting of 205,132,767 80-byte records (with 8-byte integer keys) for a total of 16,410,621,360 bytes per node. The benchmark sorts a total of 13,743,895,389 records in 46.9 minutes. The breakdown of this sort time is shown in Figure 6 and further described in the following sections.

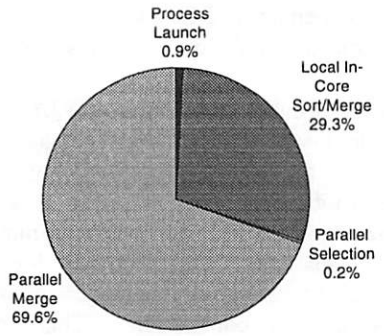


Figure 6: Breakdown of terabyte sort time

5.3.1 Process Launch and Connection Establishment

The initial 25 seconds of the sort were lost in process launching and communication initialization between sort processes. While this time may seem excessive, it was in fact the result of a significant amount of tuning. When the job starts the following must occur:

- the sort monitor must start the sort processes on each of the 67 sort nodes;
- the sort processes must establish VI connections with each of the other sort processes and the sort monitor; and finally,
- the sort monitor must send initial sort parameters to each of the sort processes.

The remote process launch was achieved with a custom RPC server on each of the sort nodes. This RPC server implemented a remote "create process" API, similar to the standard Win32 *CreateProcess* system call. Then, the nodes had to establish VI connections. Because libVI emulates a connectionless model, all VI connections must be established in advance, before the sort algorithm can start. Unfortunately, the connection mechanism in ServerNet I VI is inefficient, and highly timing dependent. While it is expected that the connection mechanism for ServerNet II will be better, connection establishment overhead remains one of the weaknesses of the VI Architecture standard, and it is unlikely it will ever be fast. If this were a significant performance problem for the sort, we could have overlapped part of the connection establishment process with Phase I of sorting. However, that would have required significant modification to libVI, which was undesirable in order to keep the code maintainable.

5.3.2 Local In-Core Sorting and Merging

Each node sorted approximately 16 GB of data in 13:05 to 13:45 minutes. As data were read and written once during sort and once during merge, 64 GB of net disk

access was accomplished at a net I/O rate of 5.3 GB/s on 67 nodes, or 78.3 MB/s per node on 7 disks spread across 4 SCSI strings on a single PCI bus. One of the stripe sets consisted of 4 7200-rpm drives on 2 SCSI strings in an external disk enclosure. The other stripe set consisted of two internal and one hot-pluggable 10K-rpm drives installed in the server. These were also spread across two SCSI strings (the drive containing the system's OS shared the SCSI string with the single hot-pluggable 10K disk). In addition to this, a few of the stripe sets used additional disks to make up for underperforming drives. As noted above, this drive count is below what is needed to saturate the SCSI buses. Peak I/O rates observed while sorting were therefore disk-count limited. Performance on this phase during stress periods was also PCI limited; with dual-PCI systems the cluster would have hit an average I/O rate of 7.5 GB/s.

The application uses multi-threading and asynchronous file operations to provide input/output concurrency during the local sort and merge phases. The 16 GB data are treated as 116 runs of about 138 MB each. Each sort process, as discussed above, maintains 30 MB of outstanding requests with 2 MB buffers. Multiple asynchronous bulk reads are issued during the sort phase, while the double-buffered merge input consists of 116 concurrent 1.25 MB requests. The merge phase is seek-count limited (which could be optimized further on large-memory systems by utilizing larger buffers and doing fewer seeks). In spite of multi-threading, the sort phase incurs occasional CPU delays.

Sort Phase	CPU Utilization (out of 2.0)
Local Sort	1.3
Local Merge	0.65
Parallel Merge	0.18-0.22

Table 1: Sort CPU Utilization

CPU utilization for the entire sort was low, and it is likely that we could have achieved similar results with uniprocessor nodes. The results are summarized in Table 1. The "local sort" was the only phase of the sort that had a CPU utilization of more than one, and the "local merge" used only ½ as much CPU as the local sort.

With the advent of Ultra2 Wide (i.e., 80 MB/sec or faster) SCSI and Fiber Channel SCSI systems, and dual, wider, and/or faster PCI busses available in many servers, we expect the time for these phases to drop below five minutes on newer systems.

5.3.3 Parallel Selection and Partitioning

Using a highly efficient parallel selection procedure, the sort took only 4-6 seconds to conduct a three round protocol of bounding, k-th record sampling, and merging, to determine what output partition needed which piece of each node's sorted data.

5.3.4 Parallel Merge

Essentially, this phase involves a global data movement in which almost every byte of data moves across the network in a random all-to-all communication pattern. This is the phase where the ServerNet I SAN and each node's PCI bus were stressed. Approximately a terabyte of data moved across the cluster in 32.6 minutes. This means that each node merged at about 8MB/sec for a total rate of 535MB/sec. To achieve this, each node had to simultaneously read its disk at 8MB/sec, write its disk at 8MB/sec, and sustain 8MB/sec of bi-directional bandwidth through its ServerNet NIC. The total bandwidth through the network was 535MB/sec. This was about 88% of the aggregate ServerNet bandwidth measured in Section 5.2, despite the increased load on the PCI bus due to SCSI disk transfers.

Because most of the real work in this phase was I/O or IPC, and both SCSI and ServerNet have very low CPU utilization, the total CPU for this phase was 0.18-0.22 across both CPUs (as shown in Table 1). While this is far less than the 2.0 CPUs we had available, we have observed that libVI's performance does drop substantially in a uniprocessor environment. This is due to the overhead associated with handling I/O and IPC interrupts and executing application code on the same processor.

The performance of this phase was PCI limited; high first-byte latencies on PCI bus drive small-transfer efficiency in to the 25% range ServerNet I's small packet size caused the inevitable PCI reads — 1 per 64-byte packet — to operate at this low efficiency.

The limiting effects of the PCI bus were further magnified by the single-threadedness of BTE. The network operated in outstanding request exhausted mode: requests could not be issued waiting for responses to earlier requests. Simulation of our topology under the Phase 3b workload [ShA99] showed that at 4 KB transfer size, a 50/50 mix of read and write requests, and a request injection rate of 4700 4KB-requests per second, the min, max and mean values of packet RTT were, respectively, 3.34, 682 and 25.8 microseconds. Therefore, whenever ServerNet I NIC went into outstanding request (OR) exhaustion (due to all 4 unacknowledged packets outstanding), performance dropped below 10 MB/sec bi-directional.

Simulations also showed that the network dynamics consisted of periodic phase transitions into and out of this OR-exhausted mode.

In addition, the 4-in 2-out topology at the periphery of the network was necessary to keep the router counts low. Output-port contention caused by traffic concentration further limited the utilization of the bisection links.

Even so, a sustained bisection bandwidth of 535MBps was achieved by the topology primarily because the asymmetric-fabric architecture kept the hop-counts low, thereby reducing the round-trip times. Many of the lessons learned about ServerNet I's large-scale performance influenced the design of ServerNet II.

With a dual-PCI server, and two ServerNet I NICs per node, the time for this phase would have been cut in half, falling to 15 minutes, without requiring any change in network topology. Coupled with 4-way 10K rpm disk striping per partition, estimated total sort time would be below 25 minutes.

With ServerNet II NICs (multi-threaded engine, efficient PCI operation due to 512-byte packet size, faster 125 MB/s links), 64-bit PCI busses, and ServerNet II's 12-port routers (which will drop inter-node distance to 2 hops), we expect Phase 3 time to drop below 10 minutes, to a point where even remote I/O will be limited by disk speeds.

To summarize, while the current result stands at 46.9 minutes to sort a terabyte of data, we envisage a time of about 15 minutes — a three-fold improvement — using commodity hardware available within the next few months. It is important to note that our sort algorithm, libPsrt and libVI are architected for scalability and efficiency; that is why, improvements in price and performance of sorting are expected to closely track forthcoming improvements in cost and performance of hardware.

6. Summary

This paper described the hardware and software used for Compaq's Sandia Terabyte Sort benchmark. The Sort exploited several key technologies including dense-racking Intel Pentium II based Compaq servers, Ultra-Wide SCSI disks and dual-channel controllers, Microsoft Windows NT 4.0, Compaq ServerNet I SAN, and the Virtual Interface Architecture. These technologies combined with a unique scalable sorting algorithm and highly efficient communication software, deliver high performance at a fraction of the cost of other solutions.

7. Acknowledgments

The authors of this paper would like to thank David Cossock, John Peck, and Don Wilson of Compaq Tandem Labs for their contributions to the sort application. Thanks to Jim Hamrick and Jim Lavallee for help with configuring and debugging ServerNet I and SnVie. In addition, we would like to thank Milt Clauser, Carl Diegert, and Kevin Kelsey of Sandia National Laboratories for purchasing the cluster, helping us put it together, and allowing us to use it for this work.

8. References

[Com97] Compaq Computer Corporation, Intel, and Microsoft, *Virtual Interface Architecture Specification, Version 1.0,* December 1997, http://www.viarch.org.

[Cos98] Cossock, D., " Method and Apparatus for Parallel Sorting using Parallel Selection/Partitioning," Compaq Computer Corporation, patent application filed November 1998.

[Fin99] Fineberg, S., Implementing a VIA-based Message Passing Library on Windows NT, Revision 1.0, Compaq Tandem Labs, January 1999.

[RiV98] Riedel, E., van Ingen, C., Gray, J., "Sequential I/O on Windows NT 4.0 — Achieving Top Performance," *2nd USENIX Windows NT Symposium Proceedings*, pp. 1-10, August 1998.

[HeG98] Heirich, A., Garcia, D., Knowles, M., and Horst, W., "ServerNet-II: a Reliable Interconnect for Scalable High Performance Cluster Computing," *Parallel Computing*, submitted for publication.

[Hor95] Horst, R. "TNet: a Reliable System Area Network," *IEEE Micro*, Volume 15, Number 1, pp. 37-45, February 1995.

[MeH99] Mehra, P. and Horst, R.W., "Switch Network using Asymmetric Multi-Switch/Multi-Switch-Group Interconnect," Compaq Computer Corporation, patent application filed March 1999.

[NyK97] Nyberg, C, Koester, C., Gray, J., *Nsort: a Parallel Sorting program for NUMA and SMP Machines*, Ordinal Technologies, Inc. white paper, November 1997, http://www.ordinal.com.

[ShA99] Shurbanov, V., Avresky, D.R., Mehra, P., Horst, R., "Comparative Evaluation and Analysis of System-Area Network Topologies," *1999 International Parallel Processing Symposium*, submitted for review.

Millennium Sort: A Cluster-Based Application for Windows NT using DCOM, River Primitives and the Virtual Interface Architecture

Philip Buonadonna, Joshua Coates, Spencer Low, David E. Culler
Computer Science Division
University of California, Berkeley
{philipb, jcoates, culler}@cs.berkeley.edu, lowtek@millennium.berkeley.edu

Abstract

We present the design and results of Millennium Sort, a distributed sorting application built using three layers of technology: extensible River System primitives, the Virtual Interface Architecture (VIA) and the Distributed Component Object Model (DCOM). The Millennium Sort application is a vehicle for exploring the issues of commercial cluster technologies and distributed development on commodity node clusters. We discuss the architecture and design of the River System primitives, VIA and DCOM. Performance results are discussed, including the latest Datamation Sort record time of 1.18 seconds achieved by a 16-node Pentium-II cluster.

1. Introduction

One of the most dynamic properties of computing clusters is the set of new technologies and methods for how to build, use and evaluate them. The relentless pursuit to "build a better cluster" offers a large design space of interesting concepts and challenging problems to tackle. A direct method to investigate such issues is to build a cluster-based application and assess the tools and technologies used behind it.

The principal goal of the Millennium Sort project is to study new commercial technologies for utilizing cluster resources and the use of commodity development tools in the context of a database oriented application (sorting). The technologies we employ include the Distributed Component Object Model (DCOM), the River System, and the Virtual Interface Architecture (VIA)[12]. Specifically, we seek to: 1) Explore the feasibility of using DCOM as a parallel remote execution system, 2) Demonstrate the extensibility of River System primitives to data management applications, 3) Investigate the use of the VI Architecture in distributed I/O systems, and 4) Evaluate the use of commodity hardware and software programming tools for distributed application development on PC clusters running Windows NT. In short, we explore the viability of building a high performance distributed system based on emerging commercial technology and available tools. This allows us to focus on component composition rather than just component design. We demonstrate an implementation of a sorting application that breaks the world record for the Datamation sorting benchmark [3] that was previously set by NOWSort [8] on the Berkeley Network of Workstations (NOW) [1].

The remainder of this report is divided into four sections, three of which detail the technical aspects of the overall system; the fourth offers a reflection on the experience as a whole. Section 2 provides background on the Windows NT cluster used for Millennium Sort and outlines the distributed sorting problem. In section 3 we present the individual component technologies of Millennium Sort (DCOM, River, and VIA) and the integration of these technologies to form the application. Section 4 details the performance of the sort in terms of the component technologies. In section 5 we discuss the project in retrospect and comment on our experience with developing a distributed application for an NT cluster. We conclude in section 6 with proposals for future work.

The 16x2 x86 Processor PC Cluster

- Dual 400Mhz Pentium II
- 256 MB SDRAM / 100Mhz Memory Bus
- 33Mhz 32-Bit PCI Bus / Ultra2 LVD SCSI
- 2x9.1GB Disk Storage
- Switched Fast Ethernet / Myrinet M2F
- Windows NT 4.0 Terminal Server Edition

Figure 1: The cluster configuration used for Millennium Sort.

2. Background

The hardware used to run the sort is a homogenous 16-node PC cluster (Figure 1) priced at ~$5800/node, including the Myrinet switch. By comparison, NOWSort used a 32-node Sun Ultrasparc workstation cluster at a cost of approximately $18,000 per-node [8]. The cluster has two principal interconnects: the Myrinet M2F System Area Network

(SAN) [15] and fast Ethernet connected through a Nortel Networks Accelar 1200 series switch. The Myrinet SAN supports the VI Architecture implementation used for the sorting application. It consists of a programmable network interface that allows emulation of VI capable hardware in a flexible system that can be instrumented. The Ethernet was used to evaluate a Winsock based version of the sort as well as provide communication services not available in the VI Architecture implementation. Since bandwidth is a primary concern with data-intensive distributed applications, we assessed the maximum and sustained throughput of various cluster node components (see Table 1). For the VI based sort, the disk is the obvious bottleneck in the system, which prompted us to stripe the two disks into a single ~23MB/sec volume on each node. For the Winsock based sort, the Ethernet becomes the limiting component.

	MEM Bus	PCI Bus	SCSI Bus	Disk	Network (Ethernet)	Network (Myrinet)
MB/sec Peak	800	133	80	21	12	150
MB/sec Sustained	640*	105*	64	14/23	10	120

Table 1: Bandwidth of cluster node components. Note that the effective disk bandwidth is actually ~23 MB/sec after striping across two disks. (* 80% estimate of peak)

The distributed sorting problem provides an aggressive space to examine and evaluate clusters. The performance of the sorting application depends on the I/O and network as well as the computational limits of the CPU. While our goal was not to conduct distributed sorting research *per se*, we use the sorting task to drive a study of the different technologies implemented around a well-known algorithm, and the cluster development tools used therein.

The Millennium Sort implements a one-pass, disk-to-disk distributed sort (Figure 2). At the start, unsorted data (keys & records) reside on the disk of each of the computing nodes. Upon invocation, each of the nodes simultaneously reads the data from the disk, partitions it according to a predetermined rule, and sends the data to the other computing nodes. The partitioning rule is typically a range of keys to a particular node. At the same time, each node receives data from the other nodes. Once all data has been read and distributed, each node sorts the data and writes the output back to disk. The total elapsed sort time is measured from the beginning of invocation (i.e. includes time to startup the application on each node) until all the data is written back to disk and the application exits. The sort is one-pass: the data set fits into available physical memory on each node of the cluster. To compare our implementation against known results, we use the Datamation sorting benchmark specification. This test measures the performance of a disk-to-disk sort of 1 million 100-byte records each with a random 10-byte key. Past results of this benchmark are available at [19]. The most recent performance record for the Datamation sort was 2.41 seconds on 32 nodes of the U.C. Berkeley NOW [8]. We demonstrate that Millennium Sort achieves twice this performance on the 16-node cluster described.

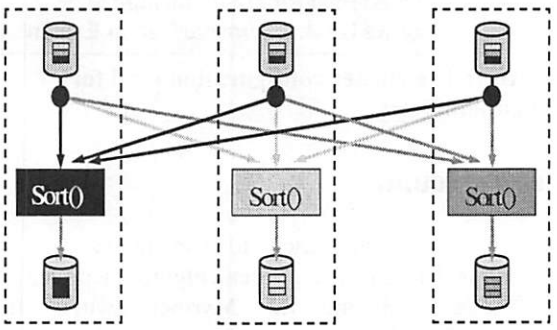


Figure 2: The generalized one-pass distributed sort algorithm.

3. Architecture and Design of Millennium Sort

The Millennium Sort application is developed around three principal technologies: Microsoft DCOM, the River System, and the Virtual Interface Architecture. In this section we describe each of these technologies and how they integrated into the sort program.

3.1. Catapult and DCOM

In order to facilitate the development and use of cluster-based applications, it is important to have some method of remote execution and job control [5]. In line with our goals of surveying industry-developed technology, we chose to use the distributed version of the Microsoft Component Object Model (COM)[4]. COM is a software development specification and a set of binary standards that allow interaction and communication amongst heterogeneous software objects. Distributed COM (DCOM) is the extension of

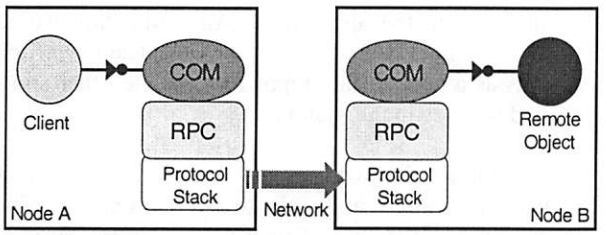


Figure 3: The DCOM Architecture.

the COM paradigm to take advantage of distributed network resources using a single programming interface (Figure 3). While DCOM is not specific to the Windows family of operating systems, Windows NT 4.0 has built in DCOM to support essential system services. For this project, we developed a DCOM based, distributed execution tool that supports invocation of generic command line programs: Catapult.

Catapult is composed of a DCOM object (a COM server executable) and a command line executable. The DCOM object is an agent, which resides on each node of the cluster and acts to service remote execution requests in a manner analogous to the UNIX *rshd*. The Catapult executable acts as the primary tool for invoking and controlling the remote programs. Prior to using Catapult, an administrative install of the DCOM agent on each node must be performed by copying the Catapult agent binary to each local disk and installing the DCOM object ClassID (a 128-bit Globally Unique Identifier) into the registry. The Catapult executable contains a command line option to do this remotely by using NT's default administrative network shares and the Remote Registry API.

To launch a distributed task, the Catapult command is issued on a user's local workstation, passing in the name of the application image and a list of nodes as arguments (e.g. `>catapult -e "hostname.exe" mm1 mm2 mm3 mm4`). For each node, the local Catapult executable spawns a new thread and requests an instance of the remote DCOM object be created. The DCOM instantiation request is received by the always-available NT RPC service running on the remote node. The RPC service references the DCOM object's ClassID in the NT Registry to find the path to the local DCOM object binary. The binary is executed (as the local user's identity) and a DCOM object is instantiated. Note that the DCOM agent binary is not actually running prior to its instantiation. The instance of the Catapult DCOM agent invokes the executable passed in at the command line. Afterwards, the agent executes a series of data methods that redirect the processes' stdout, stdin and stderr back to that node's associated thread on the user's workstation. This allows a certain amount, albeit cluttered, interactivity that is useful with certain types of remote jobs, as well as debugging distributed applications.

By using DCOM to implement Catapult, we are able to run remote executables on standard NT installations with minimal administrative overhead. Instead of requiring a separate dedicated daemon or dedicated resources, we use the NT RPC service as an *inetd*, only creating our process when called upon. Since DCOM is basically object RPC, we are able to avoid writing complex socket and stream parsing code, forgoing it for simple functions and a small amount of IDL (Interface Definition Language).

Since Catapult supports generic command line executables, the Catapult "remote execution platform" had useful applications right from its inception. An early application was to automate the already existing command line process of reloading the NIC firmware on the NT cluster. We did not require special Catapult-enhanced applications (e.g. new DCOM based applications), thus eliminating another variable in debugging sessions. A Catapult execution may be simulated through direct invocation from a system console or remote NT session, removing Catapult from the loop and allowing the use of interactive debuggers.

3.2. The River System

The River System is a distributed dataflow programming model which allows an application to *Put* and *Get* records to and from a distributed data queue. The distributed queue partitions data based on application specific rules. The River programming metaphor treats data as a fluid that flows from Sources to Sinks, which accumulate in the distributed queue of the River (Figure 4). This model is stream-based and allows direct integration with certain types of I/O intensive, distributed applications.

Distributed dataflow models that extend the stream-based metaphor of 'Data Rivers' are not a new

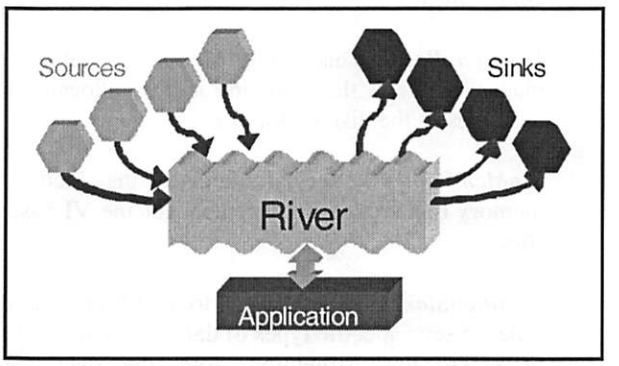


Figure 4: Basic model of the River paradigm. An application *Gets* and *Puts* data to and from a distributed queue (the River) which manages distributed source and sink data streams.

concept [2,7]. Typical distributed dataflow systems read records off of disk, partition them, either statically or dynamically across a cluster, and send the data through a specific network interface. We believe that the type, as well as the transformation applied to the data, should be orthogonal to the dataflow subsystem. This extensibility is realized through River System primitives. These primitives can be composed to describe application specific dataflows.

To understand the construction of an application using River System primitives, it is necessary to define the basic primitive types:

Source – An object that produces and possibly transforms data. Typically associated with a particular sink that the source deposits its data into.

Sink – An object that consumes and possibly transforms data. Typically receives data from a Source object or application via the SinkRecord() or SinkBuffer() methods.

Buffer – A memory object that contains a pointer to heap allocated memory.

MemPool – A data structure that manages Buffer objects, based on a simple queue structure.

These four primitives can be extended to meet specific I/O or application requirements without sacrificing the advantages of the dataflow model. For Millennium Sort these extensions included:

Disk Source & Disk Sink – Sources and sinks tailored for asynchronous disk I/O.

Net Source & Net Sink - Sources and sinks that managed network data transfers. These were implemented for both VI based and Socket based communication.

DiskMemPool – This type of MemPool creates and manages buffers that contain memory allocated in multiples of the disk sector size.

ViaMemPool - A special MemPool that included memory registration management for the VI based sort.

PartitionSink – A sink used to partition data in order to send specific types of data to specific types of sinks. For Millenium Sort, the data was partitioned lexicographically by the key in each record.

RiverSink – A sink that acts as the River System's interface to the application. An application would *Get* or *Put* records to the RiverSink. The RiverSink acts as a central data repository for the other sinks and sources in the system.

Figure 5 illustrates the overall interaction of the extended sources and sinks in Millennium Sort. The arrows indicate the logical path of data in the system. Upon activation, the system begins to first "pump" data from the Disk Sources into the River. As the River receives the data, it flows through the partitioner, which is a component of the River (1). The partitioner sends the data to its destination based on a function of the record key. Records destined for the local node get added to the queue within the River. Records that are destined for a remote node get sent to the appropriate Net Sink (2).

As this process continues, the River queue begins to fill with records from the local Disk Source, as well as incoming Net Sources (3). As the queue fills, the *Get* requests of the Sort Core empty the queue. This process continues until the Disk and Net Sources run dry, and the River queue is empty. Records are sorted and dumped back into the River with a *Put* request, which is then sunk to the Disk Sinks (4). The application waits for the Disk Sink to complete the final disk write, and then exits.

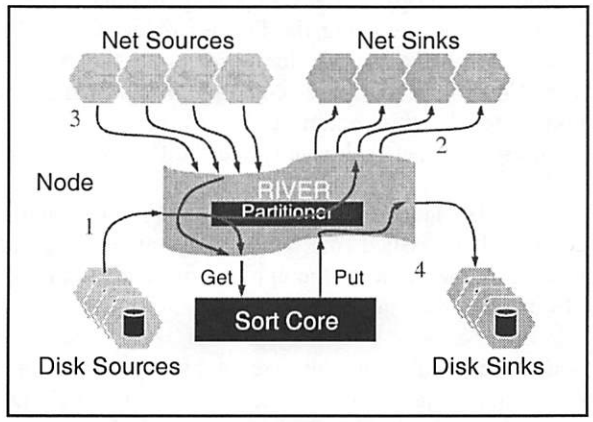


Figure 5: Illustration of dataflow in Millennium Sort.

3.3. VI Architecture

The Virtual Interface (VI) Architecture [12] is a high performance communications infrastructure that supports memory-to-memory network transfers between user processes across a cluster. The architecture is intended as a standard for user-level networking where applications have direct access to network hardware. Network resources are virtualized across user programs at the network interface level and

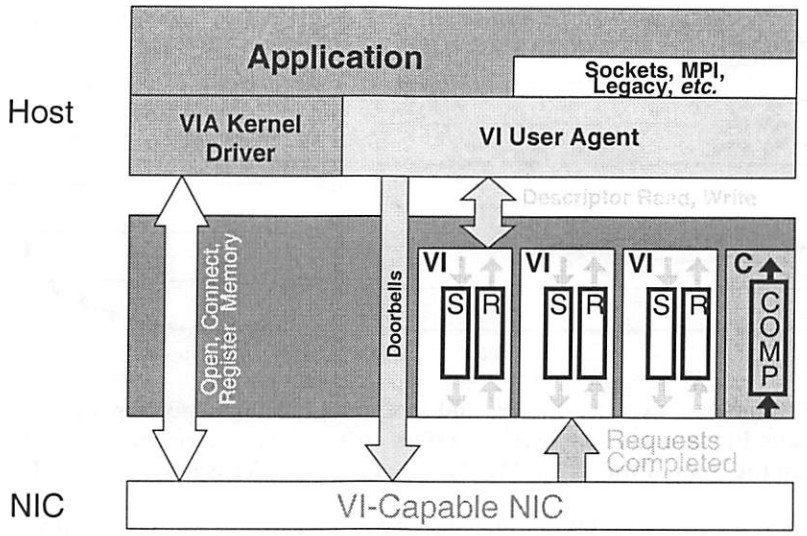


Figure 6: The Virtual Interface Architecture and its components.

OS intervention is eliminated from the critical communications path. The system has four principal components: VI Providers, VI Consumers, Virtual Interfaces, and Completion Queues (Figure 6). The design begins with the VI Provider, which includes a VI capable NIC and an OS Kernel Agent. The principal requirement for a VIA NIC is that it has resources that can be memory mapped directly into a user process address space (the doorbells). The Kernel Agent, essentially a superset of a device driver, performs the command and control functions that require operating system intervention such as device commands, memory registration and connection management. The VI Consumer component consists of the user application(s) and the User Agent, or Virtual Interface Provider Library (VIPL). The User Agent provides the API and necessary user-level support for the VI Provider implementation.

The Virtual Interface itself is the primary abstraction for the users protected, direct channel to the network hardware. Each VI consists of a pair of work queues, send and receive, their associated doorbell resources and the users registered memory regions. The work queues are a FIFO list of descriptors that mark a region of registered memory to transfer data to or from. Network data transfers are initiated by posting a descriptor in the appropriate work queue and writing a token in the queue's associated doorbell (i.e. "ring" the doorbell). The architecture supports both matched send-receive and Remote DMA (RDMA) communication semantics with either unreliable or reliable service models. When the network interface completes an operation, it sets a status mark in the descriptor that can be detected by user polling or through an event.

VI's are connection-oriented and support personalized communication with a single remote VI. This one-to-one connection model may require a large number of VIs per user to achieve full connectivity between nodes in a cluster. To provide scalable performance, Completion Queues maybe used to provide a single monitoring point for network data completions. VIs are assigned to completion queues at the granularity of the individual work queue. A completion of a descriptor places a token in the completion queue, which may be detected by either polling or through an event.

3.3.1. The Berkeley VIA Implementation

To better understand the design internals of the VI Architecture and their impact on application and cluster performance, we use our own implementation instead of an off-the-shelf product. The VIA implementation used in Millennium Sort is an enhancement of the Berkeley VIA implementation [14] that adds memory registration and increased VI/user support. The end goal of these additions is to provide a VI Architecture implementation that may be readily used by a greater variety of applications and that allows a deeper investigation into the transport itself.

The first functional component added was memory registration and virtual address translation. Prior to conducting VI communication, a user process must identify memory segments that will be used for data transfer. Memory registration locks the pages of a virtually contiguous memory region into physical memory, builds the necessary data objects to enable virtual-to-physical translations on the VI NIC, and assigns a name (memory handle) to the registered region. Our challenge was to build a registration/translation mechanism that scaled well with

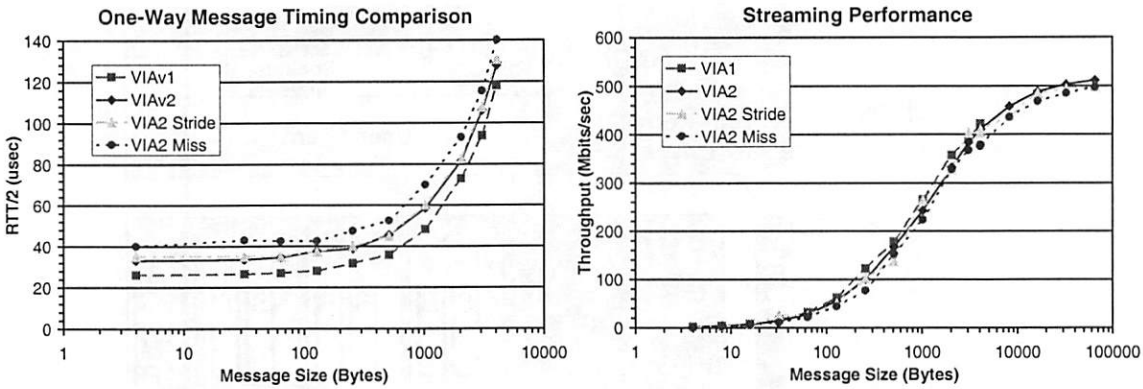


Figure 7 : One-way message timing and streaming performance for VIAv2 in comparison with the previous implementation. 'Stride' refers to the case where the test traverses a data buffer larger than the span of the TLB. 'Miss' refers to a 100% TLB miss rate in all cases.

limited resources and exhibited satisfactory performance. The result was a super-sized (1024 entry) translation lookaside buffer (TLB) on the NIC. At memory registration, the kernel agent pins the memory region into physical RAM, builds a list of the corresponding page frame numbers and passes the physical address of the beginning of the list to the network interface. The NIC stores a collection of these page list pointers in a directory maintained for each VI user process. All subsequent communication operations between the user process and NIC are conducted using user virtual addresses. Data transfers are broken into page-size segments and a TLB lookup performed for each segment. TLB misses are processed using the page directory pointer for that registered region. The benefit of this system is that memory registration scales with available host resources instead of limited NIC resources.

Expanded VI/user support was the next important functional addition to the VIA implementation. In the prototype, a memory allocation equivalent to the host page size is used to hold a single pair of doorbell registers. This mechanism was overhauled to exploit the unused space in the rest of the page. The new implementation provides 256, 128-bit doorbell pairs per page sized unit. Each doorbell page is mapped to the requesting users address space upon creation of the first VI. Since the host page size is the minimum granularity of protection for the memory system, it is possible for interference to occur between an individual user's VIs. However, any damage will be limited to that user and should be preventable if doorbell access is done through the provided User Agent.

For Millennium Sort, the VI Architecture implementation was only used to support data transfer during the sort. Other inter-node communication (i.e. DCOM calls and barriers) utilized protocols over Fast Ethernet. Other work has been done to implement DCOM over VIA which has shown to improve performance in remote method invocation [13].

4. Performance

The overall performance of Millennium Sort depends on the performance of each component. The application is built upon VIA, extended River System primitives and a parallel remote execution application based on DCOM.

4.1. VIA Performance

Assessing VIAv2 performance is important as a precursor to application performance when layered over the architecture. We found that VIAv2 functioned with only slightly degraded performance relative to the previous implementation (VIAv1) in acute measurements with no noticeable performance loss in the sort application. The performance of VIAv2 is compared against the VIAv1 prototype using one-way message timing and bandwidth benchmarks. The results are presented in Figure 7.

The one way message time is a measure of the average time it takes for a message of a given size to be transmitted from a source node and received by a remote node. It is measured by doing a series of ping-pong tests in which a message of arbitrary size is sent to a remote node that then reflects that same message back to the originator. The resulting round-trip time (RTT) is divided by two to yield the one way time. We performed this test under three principal conditions. The first was where the TLB misses only on the first use of the VI and, thereafter, exhibits a 100% hit rate. The second condition (Stride) involved a host data buffer which is larger than the address space spanned by the TLB. The benchmark traverses this buffer when making network data transfers thus, depending on the message size, forces various miss rates in the TLB. This

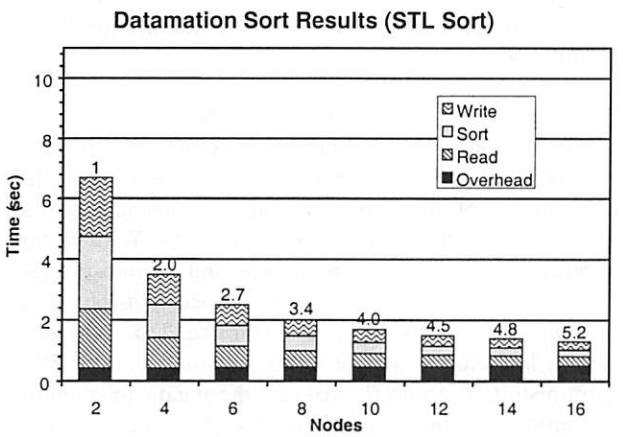


Figure 8: VI based sort performance. The numbers on top of each bar represent relative speedup.

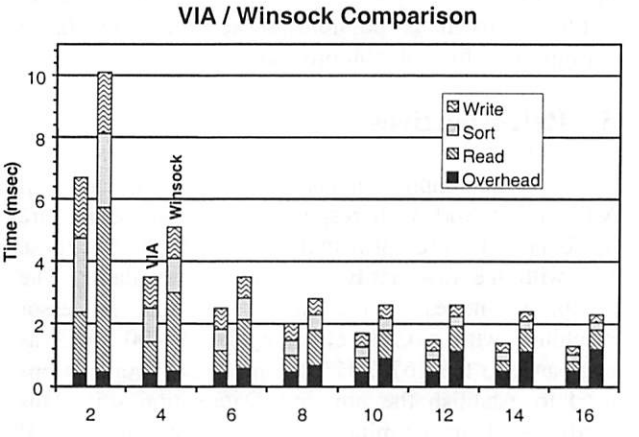


Figure 9: Side-by-side comparison of VI and WinSock based sort performance.

situation is designed to closely approximate the usage pattern of a user process transferring data to/from a large block of registered memory. The last condition (Miss) forced a 100% miss rate in the TLB. It is interesting to note that the performance difference between the large buffer and the near 100% TLB cases is minimal. This suggests that the TLB mechanism extends well to several types of network memory access patterns.

The second metric evaluated for the new implementation was streaming performance. This benchmark measures the net throughput capable when messages are sent successively from a source VI to a sink with no pause in between. We performed this test for the same three conditions as in the one-way timings. For message sizes above approximately 20KB, the throughput achieves the maximum possible value for the network interface, roughly 64MB/sec. (NOTE: while the Myrinet physical layer can achieve 150 MB/sec, the maximum rate is half this value since the VI firmware copies data to its onboard buffer before transmission). Again different TLB performance cases do not significantly impact the bandwidth of the interface.

4.2. River System Performance

The resulting performance of the River System extensions used for Millennium Sort exceeded our expectations for both the VI and Winsock based systems. Figure 8 presents the sort results for the VI Architecture based sort using the Standard Template Library qsort routine. The sort times are broken down into the major stages of the sort application: Overhead, Read and distribute, Sort and Write. The Read sequence consists of reading the data from a Disk Source, partitioning it in the River, and distributing it to the appropriate nodes through Net Sinks while simultaneously receiving data from Net Sources. The Write sequence is simply the time it takes to write data to disk via the Disk Sink. The overhead sequence is derived from the difference of total execution time of the application, from launch to completion, and time spent actually reading, sorting and writing. The overhead operations consist of: application launch via Catapult, blocking on a Winsock based barrier, and data structure/memory initialization.

On 16 nodes, the sort completed the Datamation benchmark in 1.3 seconds and achieved its best time of 1.18 seconds using a hand coded radix sort in place of the STL qsort. This record-breaking time surpassed the previous Datamation record of 2.41 seconds [8].

Figure 9 provides a side-by-side comparison of the VI based sort with the WinSock/Ethernet based sort. The overhead component is consistently higher with the WinSock implementation, increasing from 480ms to 1.2sec. This increase in overhead results from the complex mechanisms required to establish the necessary socket connections. Surprisingly, the Winsock Millennium Sort also broke the previous record with an elapsed time of 2.21 seconds (again using a radix sort core).

4.3. DCOM Performance

In Catapult, DCOM calls to a node are made from within a separate thread created for that node, both in the client and server code. In effect, all DCOM calls between nodes are asynchronous within the application as a whole. Of the overhead, approximately half is incurred by Catapult as the application scales to 16 nodes. Initial tests of a 'null' Catapult executions (startup and teardown of a 'null' process) showed times of ~180 msecs on a single node, which scaled to ~220 msecs on 16 nodes. The NT Resource Kit contains a DCOM benchmarking program, which shows that a single 'null' remote method invocation on our cluster is

approximately 400 µsecs. We believe that with further optimizations it is possible to achieve even faster startup times for Catapult processes.

5. Retrospectives

The improvements in the performance of Millennium Sort with respect to previous results are attributable to three principal causes. The first of these lies with the raw hardware resources available. The computing nodes of the cluster are dual processor machines with a CPU clock cycle of 400 MHz as compared to the 167 MHz, uni-processor workstations used to establish the previous Datamation sort. This hardware advantage minimizes time spent in the core of the sort and boosts performance of OS related communications. The second source of improvement relates to the use of the Catapult/DCOM combination as the distributed execution system. Previous distributed sorting systems spent over half the total sort time on remote invocation alone [8]. From our measurements, the DCOM system incurred startup overhead less than $1/5^{th}$ the total sort time. This improvement in remote execution alone accounts for the majority of the overall performance gain. The last source of improvement results from the use of the VI Architecture based networking. The implementation supports large message sizes (up to 100KB) with zero-copy. Additionally, the low overhead of the VI based sort contributed to better performance scaling as the number of nodes increased.

Aside from the performance results of Millennium Sort, the insight provided into the different technologies and commodity tools is extensive.

Tools. The tools used in the development and evaluation of Millennium Sort included Visual Studio 6.0 and some utilities of the NT Resource Kit. The Visual Studio application provided a clean, straightforward environment for coding and compiling, but lacked in the ability to do distributed debugging. It was not possible to connect to arbitrary remote instances of the DCOM agent or sort executable for debugging purposes. Instead, we were forced to manually debug on a collection of nodes using Terminal Server remote sessions. By contrast, the resource kit utilities were well adapted to cluster-oriented use. Most notable among these was PerfMon, which allows an administrator to view the behavior of objects such as processors, memory, cache, threads and processes. Each of these objects has an associated set of counters that provide information about device usage, queue lengths, delays, and throughput. PerfMon is capable of monitoring a collection of nodes simultaneously from a single workstation. This tool proved to be invaluable for performance analysis and debugging of the system. It is easy to use, and requires no special installation or explicit collaboration from remote nodes.

Operating System. The Windows NT 4.0 Terminal Server Edition (TSE) operating system offered an excellent environment for clustering. TSE is a multi-user version of Windows NT that provides access to a machine through remote windows sessions. With minor exceptions, this allowed us to use and administer the cluster nodes without a continuous local console. The administrative tools included with the TSE are well suited for cluster resource management. The TSE Administration program provides the means to monitor and administer node status and running processes at a cluster-wide level, allowing us to kill deadlocked jobs during debugging. TSE also includes a command line 'kill' utility that can be used in scripts to terminate a series of jobs on the cluster.

DCOM. While Catapult performed well compared to previous cluster execution systems, our overall evaluation is somewhat mixed. The DCOM based environment is robust and possesses built in mechanisms to recover from individual program crashes or global terminations (i.e. Ctrl-C). However, to maintain portability, DCOM defines a wide range of security parameters that are set on a per-machine basis. It is difficult to determine an exact setting of these parameters that would allow a wide variety of distributed applications to run without compromising security. Perhaps the most significant drawback of DCOM over Windows NT RPC is the lack of a full interactive logon to the remote machine. Credentials passed through the NT RPC service permit only a fast network or "null" logon that does not notify the Windows Networking redirector for access to remote file system volumes. There does not appear to be an interactive logon toggle available in the DCOM API. Complete Interactive logons would require the DCOM agent to be designed as an NT service running under the privileged LocalSystem account and for the user's username and password to be separately transported to the DCOM agent. Lack of redirector access requires applications invoked by catapult to be separately installed on the local disk of each node. This complicates the development process in which frequent revisions to the application occur.

VIA. The memory registration system of the VI Architecture required complicated memory management schemes within the user application. Windows NT does not allow large sections of memory to be registered in a single operation due to limited amounts of physically contiguous memory available for address translation structures. Additionally, registering

more than 80% of available physical memory yields undesirable and sometime pathologic (i.e. system crashes) operating system response. In Millennium Sort, we designed a simple windowing system that registered smaller amounts ('windows') of memory for receiving data from the network. Our results suggest that this method of registration improves application stability when registering large amounts of address space.

River. We find the most valuable aspect of the River system is the ease of extending River primitives. For instance, in order for Millennium Sort to use the VI Architecture, special operations have to be performed periodically on the memory used for buffering data. By extending the MemPool object, a VIAMemPool object is created which handles memory registration windows. This would have been difficult or impossible to integrate with a non-extensible river system without rewriting large portions of the system. In order to build the Winsock version of the application we only needed to modify the Net Source/Sink classes. These modifications required just a few hours time.

6. Future Work

The obvious next step in the Millennium Sort work is to implement a two-pass sort and continue refining our understanding of the technologies that it is composed from. A two-pass sort will allow us to run sustained sorts that will further stress test the system. Our DCOM remote execution system, Catapult, requires further performance optimizations. The extensibility of the River System through primitives worked well, but it needs to be packaged into a library, perhaps with useful primitive extensions. Lastly, although the addition of virtual memory translation to VIAv2 works and performs well, VIAv2 requires a reexamination of how it extends the memory interface to the programmer.

7. Acknowledgements

Support for this project was provided by the Microsoft Corporation, especially Jim Gray and Joe Barrera of Microsoft Research, and by Intel's Technology 2000 grant program. As well, support was provided by the National Science Foundation SimMillennium Grant (EIA-9802069), the National Science Foundation Infrastructure Grant (CDA 94-01156) and the Defense Advanced Research Projects Administration Grant (F30602-95-C-0014).

References

[1] T. E. Anderson, D. E. Culler, D. A. Patterson. "A case for NOW (Networks of Workstations)." IEEE Micro, vol. 15, (no. 1), February 1995, p. 54-64.

[2] R. H. Arpaci-Dusseau, E. A. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. A. Patterson, K. Yelick, "Cluster I/O with River: Making the Fast Case Common." to appear in *IOPADS '99*, Atlanta, Georgia, May 1999

[3] Anon. Et al. "A Measure of Transaction Processing Power." Datamation, 31(7):112-118, 1985

[4] G. Eddon, H. Eddon. "Inside Distributed COM", *Microsoft Press*, Remond, WA 1998

[5] D. P. Ghormley, D. Petrou, S. H. Rodrigues, A. M. Vahdat, T. E. Anderson. "GLUnix: A Global Layer Unix for a Network of Workstations", *Software Practice and Experience,* vol.28, (no.9), Wiley, 25 July 1998. p.929-61.

[6] E. Riedel, C. van Ingen, J. Gray, "Sequential I/O on Windows NT 4.0 – Achieving Top Performance", *Proceedings of the 2nd USENIX Windows NT Symposium*, 3-5 August 1998, Seattle, WA, pp. 1-10.

[7] T. Barclay, R. Barnes, J. Gray, P. Sundaresan. "Loading Databases Using Dataflow Parallelsim", *SIGMOD RECORD*, Vol 23, (no. 4), December 1994

[8] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, D. E. Culler, J. M. Hellerstein, D. A. Patterson. "High-Performance Sorting on Networks of Workstations." *SIGMOD '97*, Tucson, Arizona, May 1997

[9] A. C. Dusseau, K. E. Schauser, R. P. Martin, "Fast Parallel Sorting Under LogP: Experience with the CM-5." *IEEE Transaction on Parallel and Distributed Systems*, Vol. 7, (no. 8), August 1996

[10] J. Gray, J. Coates, C. Nyberg. "Performance / Price Sort", http://www.research.microsoft.com/barc July 1998.

[11] The Millennium Project: A Campus-wide cluster of clusters. University of California, Berkeley, Berkeley, CA http://www.millennium.berkeley.edu

[12]"Virtual Interface Architecture Specification. Version 1.0", *Compaq, Intel and Microsoft Corporations*, Dec 16, 1997, available at http://www.viarch.org

[13] R.S. Madukkarumukumana, C. Pu, H.V. Shah, "Harnessing User-Level Networking Architectures for Distributed Object Computing over High-Speed Networks", *Proc. of the 2nd USENIX Windows NT Symposium*, Seattle, WA, August 3-5, 1998, pp. 127-135.

[14] P. Buonadonna, A. Geweke, D. E. Culler. "An Implementation and Analysis of the Virtual Interface Architecture", *Proc. of Supercomputing '98*, Orlando, FL, 7-13 November 1998.

[15] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and Wen-King Su, "Myrinet: A Gigabit-per-Second Local Area Network." *IEEE Micro*, vol. 15, (no. 1), Feb 1995, pp. 29-36

[16] A. Basu, M. Welsh, T. von Eicken. "Incorporating Memory Management in User-Level Network Interfaces", *Hot InterconnectsV*, Stanford, CA, August 1997.

[17] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, K. Li. "VMMC-2: Efficient Support for Reliable, Connection-Oriented Communication." *Hot Interconnects V*, Stanford, CA, August 1997

[18] D. Dunning et al., "The Virtual Interface Architecture", *IEEE Micro*, vol. 18, (no. 2), Marcg/April 1998, pp. 66-75

[19] J. Gray, Sort Benchmark Home Page, http://research.microsoft.com/barc/SortBenchmark/

CPU Reservations and Time Constraints: Implementation Experience on Windows NT

Michael B. Jones

Microsoft Research, Microsoft Corporation
One Microsoft Way, Building 31/2260
Redmond, WA 98052, USA

mbj@microsoft.com
http://research.microsoft.com/~mbj/

John Regehr

Department of Computer Science, Thornton Hall
University of Virginia
Charlottesville, VA 22903-2242, USA

regehr@virginia.edu
http://www.cs.virginia.edu/~jdr8d/

Abstract

This paper presents an implementation of scheduling abstractions originally developed for the Rialto real-time operating system within a research version of Windows NT called Rialto/NT. These abstractions, *CPU Reservations* and *Time Constraints*, as described in the 1997 SOSP paper [Jones et al. 97], are intended to allow: (1) activities to obtain minimum guaranteed execution rates with application-specified reservation granularities via CPU Reservations, and (2) applications to schedule tasks by deadlines via Time Constraints, with on-time completion guaranteed for tasks with accepted constraints.

The Rialto/NT scheduler differs from the original Rialto scheduler in several key respects. First, it has been extended to schedule multiprocessors—this is the primary new intellectual contribution of this work. It has been adapted to operate with operating system clock services that only provide timing interrupts at regular periodic intervals measured in milliseconds, rather than being able to schedule clock interrupts at arbitrary sub-millisecond points of time. It coexists with the existing Windows NT scheduler, allowing it to schedule time not scheduled by itself. Finally, it has been implemented in a particularly non-intrusive manner, using rather than replacing the existing Windows NT priority-based scheduler.

Results presented will demonstrate that CPU Reservations and Time Constraints can be effectively implemented on multiprocessors. We will also describe the implementation techniques chosen and tradeoffs made as a result of implementing within Windows NT. Finally, we will present performance results and execution traces.

1. Introduction

1.1 Research Context

Novel implementations of two real-time scheduling abstractions were developed within the Rialto real-time operating system [Jones et al. 97, Jones et al. 96]: *CPU Reservations* and *Time Constraints*. These abstractions allow activities to obtain minimum guaranteed execution rates with application-specified reservation granularities via CPU Reservations, and to schedule tasks by deadlines via Time Constraints, with on-time completion guaranteed for tasks with accepted constraints.

The goal of this work is to investigate the feasibility of bringing benefits of predictable Rialto-style scheduling to Windows NT applications. This paper describes a re-implementation of the Rialto scheduling abstractions in a research version of Windows NT called Rialto/NT. Our implementation is based on Windows 2000 Beta 3.

This paper assumes that the reader is already familiar with the results and techniques presented in [Jones et al. 97] and builds directly upon them.

1.2 New for Windows NT

There are several key differences between Rialto and Windows NT that affected this work. Some of these are:

- **Multiprocessor** — Windows NT can be run on symmetric multiprocessors. The Rialto scheduler was designed only for scheduling uniprocessors.
- **Periodic Clock** — Time is kept on Windows NT using periodic interrupts that advance the system's record of the current time. The interrupt frequencies are settable to values supported by the Hardware Abstraction Layer (HAL) being used; however, these values are restricted to integer multiples of milliseconds. (For more on HALs and timing see [Jones & Regehr 99].) On Rialto, the clock interrupts occurred on an aperiodic, as-needed basis with precision on the order of one microsecond.
- **Existing Scheduler** — Windows NT has an existing priority-based scheduler. Under Rialto, ours was the only scheduler. One of the Rialto/NT goals is to coexist with the existing Windows NT scheduler, allowing applications using it to obtain approximately the same behaviors as they did before our changes.

A related decision that distinguishes this work from both the original Rialto implementation of CPU Reservations and Time Constraints and the previous Vassal [Candea & Jones 98] scheduling work on Windows NT is that we decided to implement the Rialto/NT scheduler by taking advantage of, rather than circumventing, the existing Windows NT priority-based scheduler.

A final distinction between the current Rialto/NT implementation and the original Rialto system is that, as of this writing, we have not yet fully implemented the *Activity* abstraction. Consequently, our CPU reservations currently apply to a specific thread, rather than to all threads within an activity. We view this as an interim implementation step—not a long-term design decision.

2. Programming Model

2.1 Adaptive Real-Time Applications

The Rialto scheduling abstractions were designed to allow multiple independently authored applications to be concurrently executed on the same machine, providing predictable scheduling behavior for applications with real-time requirements. They were designed to enable applications to perform predictably in dynamic, open systems, where such factors as the speeds of the processor, memory, caches, busses, and I/O channels are not known in advance, and the application mix and available resources may change during execution.

Applications with real-time requirements in such a dynamic environment cannot rely on off-line schedulability analysis, unlike those for single-purpose systems with fixed hardware configurations and application loads. Consequently, real-time applications must monitor their own performance and resource usage, modifying their behavior and resource requests until their performance and predictability are satisfactory. The system plays two roles in this model. It provides facilities both for applications to monitor their own resource usage and for applications to reserve the resources that they need for predictable performance.

2.2 Terminology and Abstractions

Two additional abstractions are provided in Rialto/NT beyond those provided in the normal Windows NT system: *CPU Reservations* and *Time Constraints*. This section is intended to provide a brief introduction to them and their usage for those unfamiliar with them.

2.2.1 CPU Reservations

CPU Reservations are made by threads to ensure a minimum guaranteed execution rate and granularity. CPU reservation requests are of the form *reserve X units of time out of every Y units for thread A*. This requests that for every time interval of size Y, thread A be scheduled for at least X time units, provided it is runnable. For example, a thread might request at least 800µs every 5ms, 7.5ms every 33.3ms, or one second every minute.

CPU Reservations are *continuously guaranteed*. If A has a reservation for X time units out of every Y, then for *every* time T, A will be run for at least X time units in the interval $[T, T+Y]$, provided it is runnable.

Blocked threads do not accumulate credits for time reserved but not used; unused time is given to other threads that are ready to run.

In Rialto, CPU Reservations applied to *Activities*, which were sets of threads, rather than just individual threads. We plan to eventually augment the Rialto/NT implementation with activities as well.

2.2.2 Time Constraints

A ***Time Constraint*** is a dynamic request issued by a thread to the scheduler that the code associated with the constraint be run to completion between the associated start time and deadline. The request also contains an upper bound on the execution time of the code.

Feasibility analysis is done for all time constraints when submitted, including those with a start time in the future. The requesting thread is either guaranteed that sufficient time has been assigned to perform the specified amount of work when requested or it is immediately told via a return code that this was not possible, allowing the thread to take alternate action for the unsatisfiable constraint. For instance, a thread might skip part of a computation, temporarily shedding load in response to a failed constraint request. Providing time constraints that can be guaranteed in advance, even when the CPU resource reservation is insufficient or non-existent, is one feature that sets Rialto and Rialto/NT apart from other constraint- and reservation-based schedulers.

When a thread makes a call indicating that it has completed a time constraint, the scheduler returns the actual amount of execution time the code took to run as a result from the call. This provides a basis for computing accurate run-time estimates for subsequent executions.

An application can request that a piece of code be executed by a particular deadline as follows:

```
Calculate constraint parameters
schedulable = BeginConstraint(
    start_time, estimate, deadline);
if (schedulable) {
    Do normal work under constraint
} else {
    Transient overload — shed load if possible
}
time_taken = EndConstraint();
```

The *start_time* and *deadline* parameters are straightforward to calculate since they directly follow from what the code does and how it is implemented. The *estimate* parameter requires more care, since predicting the run time of a piece of code is a hard problem (particularly in light of variations in processor & memory speeds, cache & memory sizes, I/O bus bandwidths, etc., between machines) and overestimating it increases the risk of the constraint being denied.

Rather than trying to calculate the *estimate* in some manner from first principles (as is done for some hard real-time embedded systems), one can base the estimate on feedback from previous executions of the same code. In particular, the *time_taken* result from EndConstraint() provides the basis for this feedback.

The *schedulable* result informs the calling code whether a requested constraint can be guaranteed, enabling it to react appropriately when it cannot. This might be caused by transient overload conditions or an application optimistically trying to schedule more work than its CPU reservation can guarantee.

A composite EndConstraint/BeginConstraint call that atomically ends the previous constraint and begins a new one is also provided.

Finally, note that constraint deadlines may be small relative to their thread's reservation period. For instance, it is both legal and meaningful for a thread to request 5ms

of work in the next 10ms when its reservation only guarantees 8ms every 24ms. The extra time is guaranteed, when possible, using free time in the schedule. The request may or may not succeed, but if it succeeds sufficient time will have been reserved for the constraint.

3. Implementation

3.1 Precomputed Scheduling Plan

The principal data structure for Rialto/NT is the *Precomputed Scheduling Plan* [Jones et al. 97], a tree-based representation of time that allows the system to efficiently schedule reservations with a wide range of periods, from a few milliseconds to tens of seconds, and to decide in constant time what to schedule next. We maintain a scheduling plan for each processor in the system.

Nodes in the scheduling plan represent either intervals of time assigned to activities with CPU reservations or free intervals. Attached to nodes are lists of *interval assignments*, which represent time intervals reserved for threads with constraints.

While Rialto/NT maintains a scheduling plan for each processor in the system, it does not currently ensure that reserved time is scheduled on any particular processor. Rather, it uses the Windows NT scheduler's priority scheduling to dispatch threads. We discuss this implementation decision in more detail in Section 3.5.2.

3.2 Policy Decisions

A multiprocessor implementation of Rialto's scheduling abstractions is necessarily more complex than a uniprocessor implementation. Although most of the mechanisms changed only slightly, the space of policy choices increased dramatically.

3.2.1 CPU Reservation Policies

As a simplifying assumption, we decided that once a reservation is assigned to a scheduling plan, it stays there for its lifetime. When a new reservation is requested, the system attempts to add it to the scheduling plans in which it could possibly fit, in increasing order of CPU utilization. Threads may not request a reservation in a specific plan or on a specific processor. If a reservation cannot be added to any scheduling plan, the request fails. Clearly, there are situations where this scheme rejects a reservation that could have been granted by redistributing reservations among plans and rebuilding all plans at once. We chose not to do this for now, as rebuilding an individual scheduling plan is already potentially time-consuming. However, a global rebuild would give us more freedom to implement optimizations such as placing reservations with similar periods in the same scheduling plan, helping to minimize nonessential context switches.

To rebuild a scheduling plan, Rialto/NT begins with an empty plan and adds reservations in order of increasing period. In combination with a search strategy that backtracks when it cannot fit all reservations into the plan, this ordering tends to achieve a high percentage of processor utilization. The use of a heuristic search is critical, as the problem of optimal reservation layout is NP-complete, even for a single processor.

Each CPU can have an *idle reservation*, a reservation for no thread. Since this time is scheduled by the existing Windows NT scheduler, the idle reservation prevent Rialto/NT from starving user interface or worker threads, no matter how many reservations and constraints exist.

3.2.2 Time Constraint Policies

New time constraints are placed in the scheduling plan in which the requesting thread has a reservation, if any. Otherwise, scheduling plans are presently tried in numerical order. A better heuristic in this case might be to try plans in order of increasing load.

3.3 Entry Points

We added four system calls to Windows NT:

```
NTSTATUS NTAPI NtBeginReservation (
    IN HANDLE ResThread,
    IN ULONG Period,
    IN ULONG Amount,
    OUT ULONG *ActualPeriod,
    OUT ULONG *ActualAmount,
    OUT ULONG *Cpu);

NTSTATUS NTAPI NtEndReservation (
        IN HANDLE ResThread);

NTSTATUS NTAPI NtBeginConstraint (
    IN _int64 Start,
    IN _int64 Deadline,
    IN _int64 Estimate,
    IN BOOLEAN EndPrev,
    OUT _int64 *TimeTaken,
    OUT ULONG *Cpu);

NTSTATUS NTAPI NtEndConstraint (
    OUT _int64 *TimeTaken);
```

Due to rounding and quantization effects, NtBeginReservation() may not be able to grant the precise amount and period requested. So, it returns to the caller the actual amount and period of the reservation granted. What is guaranteed is that the actual reservation period is less than or equal to the requested period and that the fraction of the CPU granted is at least as large as the fraction requested.

Both reservation and constraint requests, if successful, report the scheduling plan that the request was granted on. This is a debugging aid, and may later be removed.

As well as being called by applications, the Rialto/NT scheduler is also invoked via kernel callback routines and Windows NT timers.

3.4 Use of Windows NT Timers

There is a Windows NT timer associated with each scheduling plan. Timer callbacks are set for times when the scheduling plan needs to schedule a different thread.

Windows NT keeps times internally as 64-bit quantities in 100ns units. Although some CPUs and interrupt controllers can provide very high resolution times to user programs, the most precise time used by Windows NT itself is *interrupt time*, which is advanced by

the clock interrupt handler. Windows NT timers, set using NtSetTimer(), expire at a certain interrupt time; the clock interrupt handler scans a list of timers, queuing a *Deferred Procedure Call* (DPC) for each expired timer. The DPCs perform the work associated with the timers after the clock interrupt handler finishes. The clock interrupt period typically defaults to 10-15ms, depending on the HAL. To support applications that need more fine-grained timing, many HALs support variable clock interrupt periods in 1ms increments. HALX86, the default x86 HAL, supports periods down to 1ms; similarly, HALMPS, the default multiprocessor HAL, supports periods down to 1/1024s. (The real time clock, which HALMPS uses, only supports clock periods that are power-of-two divisions of a second.)

To make the best possible use of discrete interrupt times, Rialto/NT is designed so that CPU rescheduling (transitions between nodes of a scheduling plan) always occurs at the time that a clock interrupt is delivered. This eliminates further rounding errors. Typically, we set the interrupt period to 1ms before initializing Rialto/NT so as to better schedule reservations with small periods.

3.5 Scheduling Threads

Our first attempt at a mechanism for scheduling threads was intrusive and low-level; problems with this approach led us to scrap it for an indirect, less intrusive method. For brevity we will occasionally refer to a thread scheduled by the Rialto/NT scheduler as an *RT* thread.

3.5.1 Initial Implementation

We initially added code to the clock interrupt handler to check if Rialto/NT needed to make a scheduling decision, and if so, to call our decision code. We also modified the dispatcher return path to see if there was a thread that our scheduler had decided to run. If so, it annulled whatever decision the Windows NT scheduler had made by putting the standby thread back on a ready list and replacing it with the RT thread, which then ran immediately. Although this approach had the advantage of performing scheduling at a very low level, it had several disadvantages. It violated the principle of localized cost by adding code to frequently used code paths, imposing a performance penalty on threads not using the real-time subsystem. It also let the Windows NT scheduler do the work to make a scheduling decision, and then often ran a different thread. (The alternative to this, preventing the Windows NT scheduler from running unless Rialto/NT had nothing to decide, was even more intrusive.)

Locking issues caused a subtler problem; we wanted to use a spinlock to protect Rialto/NT data structures, but our code in the dispatcher was called with the *dispatcher database lock* held—this lock protects all Windows NT scheduler data structures. Then, if we acquired our own spinlock, we would have forced ourselves to never acquire these two locks in the other order since that risks deadlock. This was an impossible restriction, because many of the Windows NT kernel functions that we wanted to call from our code, with our lock held, acquire the dispatcher database lock. So we used the dispatcher database lock to protect both the Rialto/NT scheduler and the Windows NT scheduler. Unfortunately this not only increased contention for an already busy lock, it also made programming inconvenient since the kernel memory allocation functions ExAllocatePool() and ExFreePool() cannot be called with the dispatcher database lock held. We were forced to pre-allocate memory before acquiring the lock and to defer frees until it was released. Together, these problems were serious enough that we decided to use a different means of scheduling threads.

3.5.2 Use of Priority Scheduling by Rialto/NT

Windows NT has 32 priorities [Solomon 98, p. 187]. 0 is reserved for the zero page thread. 1-15 are for time-sharing threads, which are subject to increased quanta for threads in the foreground process and priority boosts under certain circumstances [Solomon 98, p. 205]. Priorities 16-31 are "real-time" priorities; Windows NT never adjusts the priorities or quanta of threads in this priority range and simply schedules among runnable threads at the highest priority in a round-robin manner.

The current Rialto/NT implementation schedules threads using the Windows NT scheduler, rather than bypassing it. To schedule a thread, we raise it to priority 30. Obviously, for this method to work, no thread outside of Rialto/NT may spend significant amounts of time running at priority 30 or 31.

Rescheduling employs the following steps: A clock interrupt occurs and our DPC is enqueued; it runs and lowers the priority of the currently scheduled RT thread from 30 to its previous value. Then, the scheduler selects the next node in the scheduling plan, saves the priority of the thread corresponding to that node, boosts it to 30 using KiSetPriorityThread(), sets a timer to expire at the end of the node's time slice, and exits. The Windows NT scheduler then dispatches the thread selected by Rialto/NT and it begins running.

We made one small change to KiSetPriorityThread(). When it is called from outside of Rialto/NT, we need to check if Rialto/NT is currently scheduling the thread whose priority is being adjusted. If so, we modify the saved priority rather than the actual one; the thread will be set to the new priority when it is descheduled.

Because Rialto/NT schedules threads by manipulating Windows NT priorities, it does not matter if a thread that is being scheduled blocks, as CPU guarantees just apply to runnable threads. The only thread state changes that matter are changes to reservations and constraints, which are made via new system calls, and thread termination, of which Rialto/NT is notified by a callback set during initialization using PsSetCreateThreadNotifyRoutine().

The low intrusiveness of our scheduler gives us confidence that it could easily be made into a Vassal-style loadable kernel module [Candea & Jones 98]. We are currently researching loadable scheduler interfaces.

3.5.3 Multiprocessor Issues

We initially considered pinning scheduling plans to processors by manipulating the *affinity masks* of RT

threads. Affinity masks are attributes of Windows NT threads that restrict them to be scheduled only on a subset of the available CPUs. However, pinning RT threads to a single CPU would prevent them from opportunistically using free time on other processors. So, Rialto/NT instead allows the Windows NT scheduler to decide on which processor to run RT threads, depending upon its processor affinity logic and scheduling plan induced priority scheduling to keep threads running on the same CPU, so as to minimize inter-processor cache traffic.

Because the number of scheduling plans is the same as the number of CPUs, we assumed that there would never be contention for processors among threads at priority 30. However, this is not the case: there are situations on multiprocessors when Windows NT *does not schedule the highest-priority runnable threads* [Solomon, pp. 213-215]. A relevant situation is the selection of a processor on which to run a newly ready thread. In the absence of idle processors, Windows NT picks a processor and preempts the thread running on it only if that thread's priority is less than the priority of the new thread. Otherwise, the new thread is added to a ready list and does not get to run immediately. Because only a single processor is considered, this scheme misses the case where a thread of lower priority is running on a different CPU. As shown in Figure 4-7, this case can be quite common.

To cause Windows NT to always schedule ready RT threads, we modified the processor selection logic for these threads in KiReadyThread() to consider preempting the thread running on each processor in the affinity mask of the newly ready thread until the preemption is successful or all processors have been tried. Figure 4-8 shows the results of the improved code. KiReadyThread() and KiSetPriorityThread() were the only two kernel functions that we modified while implementing Rialto/NT.

3.6 Concurrency Control

The Rialto/NT data structures are protected by a single spinlock; we could fairly easily change this to one spinlock per scheduling plan. Breaking up the locks more than that is unlikely to be practical or profitable.

Rialto/NT still sometimes acquires the dispatcher database lock, but only briefly while adjusting thread priorities. The lock ordering we have chosen prohibits the Rialto/NT spinlock from being acquired when the dispatcher database lock is held.

Part of the process of acquiring a reservation involves a search with backtracking. When there are many reservations, this can take several milliseconds even on a fast machine. This is far longer than the 25µs maximum recommended spinlock hold time [Microsoft 99, sec. 16.2.5], so Rialto/NT uses a form of optimistic concurrency control to avoid holding the lock during this potentially long computation. The plan building routine makes copies of the relevant data with the spinlock held and also records a version number associated with the scheduling plan. It then releases the lock and builds a new plan. (Because the Windows NT kernel is fully reentrant, it is not harmful for threads to spend a long time running in kernel mode.) When the plan is finished, Rialto/NT reacquires the spinlock and checks the version number. If it has changed, the new plan is useless and must be discarded; otherwise the scheduler swaps the new and old scheduling plans, increases the version number, releases the lock, and deallocates the old plan. Every routine that modifies a scheduling plan must be careful to increment the version number before releasing the lock.

3.7 Damage Control

Rialto/NT runs at a high level for a scheduler. Unfortunately, this means that without correct and timely behavior from lower-level portions of Windows NT, some of the guarantees that it makes will not be met. However, the code simplification achieved using the high-level approach is significant; indeed, without major design changes to Windows NT, many lower-level thread scheduling approaches would not do much better. Furthermore, because it is written as high-level processor-independent code, Rialto/NT should be trivially portable between CPU architectures.

3.7.1 Late DPCs

The most vulnerable part of the scheduler is the timer DPC that schedules RT threads. When a DPC is queued, the kernel requests a software interrupt; this interrupt will not occur until the *interrupt request level* (IRQL) goes below DPC level. Therefore, DPCs may be prevented from running by interrupt handlers, by threads running in the kernel at elevated IRQL (while holding a spinlock, for example), and by other DPCs. Our experience [Jones & Regehr 99] shows that other DPCs are the main problem and that by carefully choosing which device drivers run on a system, long-running DPCs can be minimized.

Even so, the scheduler DPCs will be called late sometimes. When this occurs, Rialto/NT minimizes the damage to the overall schedule by penalizing the threads that the DPC would have scheduled if it had run on time. Our goal is keep the scheduling plan on time at all costs. To this end, we keep a virtual time for each scheduling plan, which remains synchronized with real time as long as the scheduler DPC is called on time. When the virtual time is lags behind, it means that our code was called late and Rialto/NT catches up by walking the scheduling plan forward until the times are again in synchrony.

3.7.2 Interrupt Time Skew

The interrupt period supplied by HALs to the kernel is not always completely accurate. To prevent accumulated round-off error, the clock interrupt handler may not always add the same value to the interrupt time. This prevents Rialto/NT from making accurate predictions about the correspondence between future interrupt times and wall clock times, and therefore it cannot guarantee that constraint start times and deadlines will be honored. We currently ignore the problem since only certain HALs perform this correction, and the worst possible drift under HALMPS amounts to 9ms per hour. There is no problem under HALX86—it always adds a constant to interrupt

time. A solution to this problem would most likely involve exposing and taking into account the HALs' (simple) round-off error avoidance logic. We cannot just have the HAL tell the scheduler the value that it adds to the interrupt time—it needs to know the values that will be added in the future.

3.7.3 Lost Clock Interrupts

The only notion of passing time that Rialto/NT currently understands is interrupt time. If interrupt time fails to progress because clock interrupts are missed (for example, when the PCI bus is blocked by a write to a full FIFO on a video board [Jones & Regehr 99]), the scheduling plan will slip with respect to real time. We have not handled this case because only the most egregious hardware/driver combinations cause clock interrupts to be missed. In the future, we could handle this case by detecting the missed interrupts using the Pentium timestamp counter and then catching up.

3.8 Execution Time Reporting

Because Windows NT thread execution time accounting does not provide millisecond accuracy, we are not yet giving threads precise feedback on their time taken during constraint execution. It would not be sufficient for just Rialto/NT to provide accurate accounting, both because it is unaware of blocking threads and because the Windows NT scheduler may also schedule RT threads. We are investigating ways to provide accurate accounting.

4. Results

4.1 Experimental Setup

All performance results reported were measured on a Gateway E-5000 dual-processor 333 MHz Pentium II PC with 128MB of memory. Although the machine normally uses both processors, it is also possible to tell Windows NT to use only one processor by using the /numproc=1 switch in c:\boot.ini. Uniprocessor measurements were collected in this way.

The machine uses an Intel EtherExpress Pro/100B PCI Ethernet adapter, an Adaptec AHA-3940U/UW dual SCSI controller, and a Seagate ST10101W SCSI disk.

The current version of Rialto/NT is based on Windows 2000 Beta 3. Our build is not as highly optimized as the released binaries, which makes some kinds of debugging easier, albeit at a cost in performance.

All time measurements were made in user space with the Pentium timestamp counter. Times include all overheads, such as the time to enter and leave the kernel.

4.2 Size Results

The Rialto/NT scheduler contains about 6000 lines of C, which are divided roughly equally between reservations and constraints. Maximum dynamic scheduler memory usage is under 100KB during simulation runs with many reservations and constraints (as per Figures 4-2 and 4-3).

4.3 Micro-Benchmarks

Figure 4-1 demonstrates the additional context switch overhead introduced by Rialto/NT's scheduling

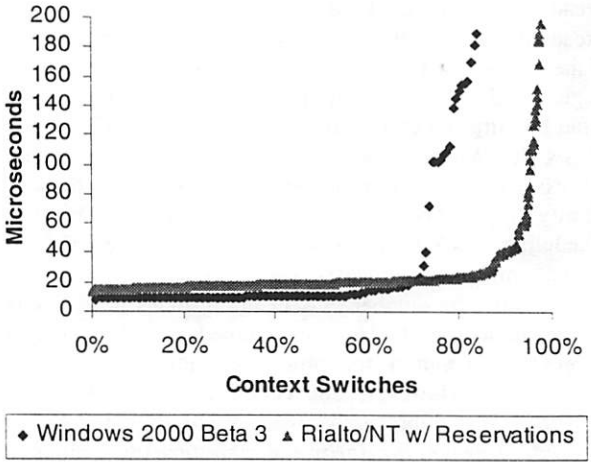


Figure 4-1: Rialto/NT vs. native context switch times

implementation. During measurements, the system was booted in single-processor mode and ten threads were competing for the CPU. In one case, the threads were run under the released version of Windows 2000 Beta 3 and in the other, they were scheduled by Rialto/NT with reservation amounts between 2ms and 18ms, all with period 128ms. The released Windows NT kernel has a minimum context switch time of 8.4µs, with a median of 10.6µs. Rialto/NT has a minimum of 13.6µs and a median of 18.6µs. So, we conclude that reschedules performed by the Rialto/NT mechanism add approximately 8µs to the context switch time. The minimum scheduling quantum on Windows NT is approximately 1ms, so this represents at most a 0.8% overhead.

However, two data sets that are not shown here (the context switch times for an unmodified version of the Windows NT kernel that we rebuilt and for the Rialto/NT kernel without any reservations) show nearly identical context switch times that are around 2.2µs slower than the released kernel. We believe that this is because our Rialto/NT build is not as highly optimized as the released kernel. Hence, we would expect context switches in a fully optimized build of Rialto/NT to be at least 2µs faster than the results presented here. Finally, note that the larger Rialto/NT context switch times are squeezed to the right-most part of the graph. This is because the CPU reservations forced many more context switches to occur under Rialto/NT than did under Windows NT.

Figure 4-2 graphs the times to make an intentionally complex cumulative set of CPU reservations. All requests reserve 1ms but at varying periods. The sequence of periods is a pattern which begins 4s, 4s, 2s, 4s, 2s, 1s, 4s, 2s, 1s, 0.5s, etc. This sequence was chosen to build as complex and sparse a scheduling graph as possible, allowing us to measure what we believe to be worst-case times. Both single- and dual-processor times are reported.

The dual-processor reservation times are approximately half those of the uniprocessor. This is because reservations are split between the two per-processor scheduling plans, each of which is

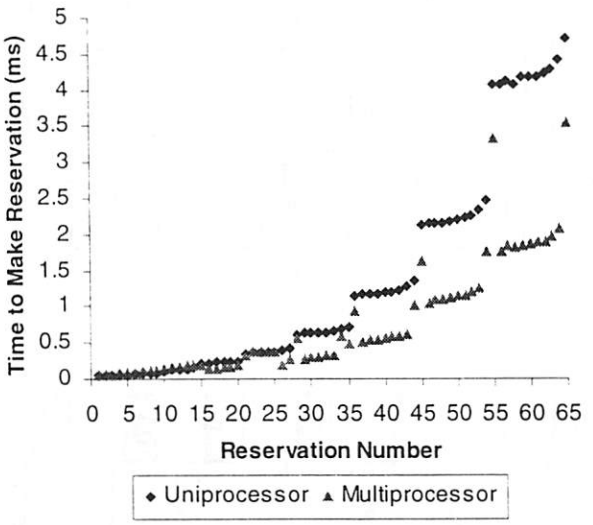


Figure 4-2: Times to make simultaneous reservations

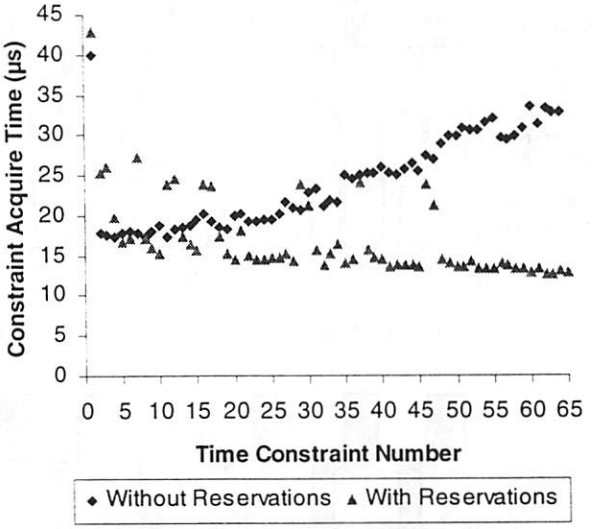


Figure 4-3: Times to begin simultaneous constraints

approximately half the size of the corresponding uniprocessor plan.

While the maximum reported values of approximately 5ms are significant, it should be noted that the times to make the first 35 reservations are all below 1ms and the first 7 are all below 100µs. Indeed, we believe a small number of reservations to be the common case. The X-axis of the graph represents the number of simultaneous *independent* granted CPU reservations. To reach the ~5ms values, one would have to have 64 simultaneous real-time applications on the same machine, meaning that the average application is content with less than 1.6% of the CPU. Yet even for this very unlikely case, these unoptimized reservation acquisition times are still reasonable, given that reservation requests will typically occur infrequently, normally just at program startup or at major mode changes.

One comparison with Rialto for this experiment is warranted. In Figure 5-1 in [Jones et al. 97], which corresponds to this experiment, many of the reservation times are near zero. This is because instead of rebuilding the entire scheduling graph for each new reservation, Rialto first looks for a set of free slots large enough to accommodate the new reservation. If they exist, Rialto incrementally adds the new reservation to the existing graph. This is an optimization we have coded but not yet tested and enabled on Rialto/NT.

Figure 4-3 graphs the time to begin simultaneous time constraints in two cases. One case is for a system with no active CPU Reservations. The other is for a system with reservations as in Figure 4-2. The no-reservations case shows slow linear growth in time with the number of pending constraints, as the constraint acquisition code is forced to search farther ahead in the plan to find free time. The times for acquiring constraints in the case of threads with reservations shows no such increase because the scheduling plan data structure allows the constraint acquisition code to examine only times reserved for the thread and unreserved times, never considering times reserved for other threads. We believe that the longer time to acquire the first constraint is due to cache effects.

A possibly more useful measure of constraint speed is the amount of time for the atomic operation that ends the previous constraint of a thread and begins a new one. This would typically be employed in a loop. We measured this cost for a thread with no reservation in six runs of 150 loops each. In a typical run, the minimum, median, and mean times are very close together, respectively 8.2µs, 8.3µs, and 8.4µs, with a maximum of only 24.8µs.

4.4 Scheduling Traces

This section shows a number of scheduling traces taken on single- and dual-processor boots of Rialto/NT.

Figure 4-4 shows an execution trace on a single-processor boot of three threads with reservations of differing amounts and periods competing with a high-priority thread. (The high-priority thread is set to a high priority within the time-sharing class. This is lower than the real-time class priority used by Rialto/NT to schedule threads.) The actual amounts and periods of the reservations differ from the requested amounts and periods: the thread requesting 1ms/10ms was granted 1ms/8ms, the thread requesting 4ms/20ms was granted 4ms/16ms, and the thread requesting 16ms/40ms was granted 13ms/32ms. Because the high-priority thread runs whenever no thread has a CPU Reservation, one can clearly see the regular nature of the reservations; threads 1, 2, and 3 only run during their reserved times.

Figure 4-5 shows an execution trace like that in Figure 4-4 except that thread 3, which has a reservation of 1ms every 10ms, also uses a Time Constraint after each 2ms of its own execution to request 2ms of CPU time in the next 10ms, effectively doubling its amount of CPU for the next 10ms period when the constraint is accepted. Thread 3's constraints do typically succeed in obtaining

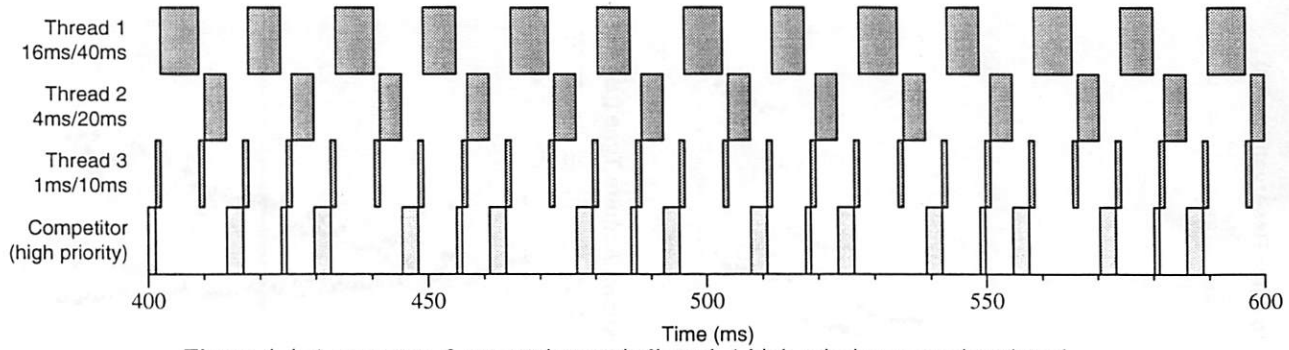


Figure 4-4: 1 processor, 3 reservations as indicated, 1 high-priority competitor thread

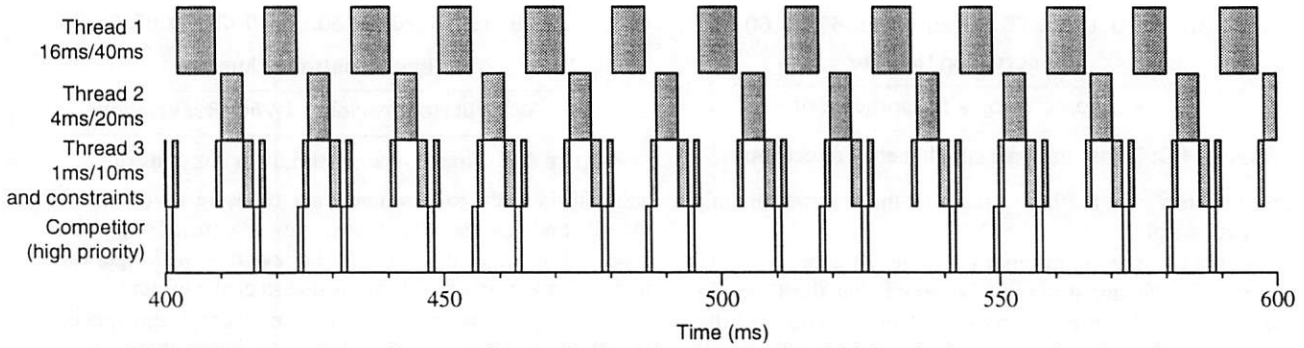


Figure 4-5: 1 processor, 3 reservations as indicated + 1 constraint, 1 high-priority competitor thread

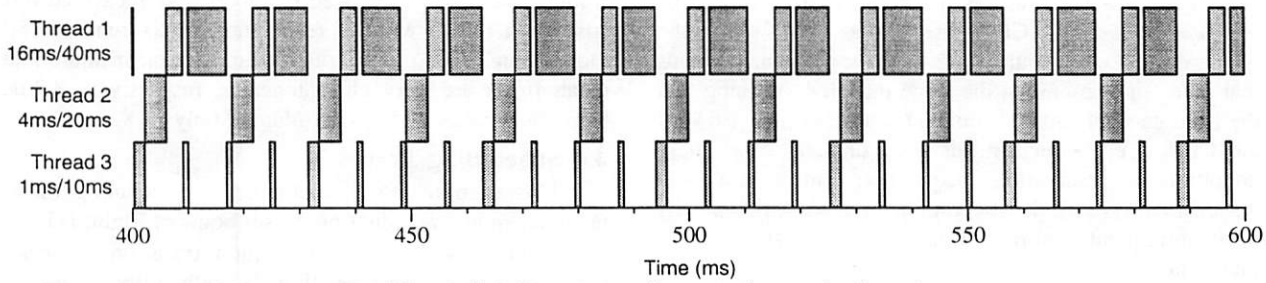


Figure 4-6: 1 processor, 3 reservations as indicated

the additional time despite the high-priority competitor thread.

Figure 4-6 shows an execution trace like 4-4 except that there is no competitor thread. Threads 1 and 3 both are scheduled by the default Windows NT scheduler at times other than during their reservations, while still being scheduled during their reservations by the Rialto/NT scheduler. It is not clear to us why the Windows NT scheduler never chooses thread 2. This is an example of the default scheduler not allocating unreserved time "fairly" between threads with reservations of differing periods. In contrast, the Rialto scheduler achieved fairness by scheduling unreserved time itself.

Figures 4-7 and 4-8 show execution traces taken on dual-processor boots. The reservation periods are the same as in Figure 4-4, but the amounts are doubled. The thread requesting 2ms/10ms was granted 2ms/8ms, the thread requesting 8ms/20ms was granted 7ms/16ms, and the thread request 32ms/40ms was granted 26ms/32ms. Consequently, more than 100% of a single CPU is reserved. To allow threads 1, 2, and 3 to run only during their reservations we employed two high-priority competitor threads.

Figure 4-7 shows a run on a kernel without the modification to KiReadyThread() we described in Section 3.5.3 that forces Windows NT to always run an RT thread when it becomes ready. For example, thread 3 should have been scheduled at about 450ms into the run. Because it was raised to priority 30 and still did not run, we infer that the default Windows NT KiReadyThread() routine tried to schedule thread 3 on the same processor that thread 1 was already running on. Thread 1 was not preempted because it was also running at priority 30, and consequently thread 3 was put onto a ready list instead of being scheduled. Figure 4-8 shows a run on a kernel containing the modified version of KiReadyThread(). Because the threads running on both processors were

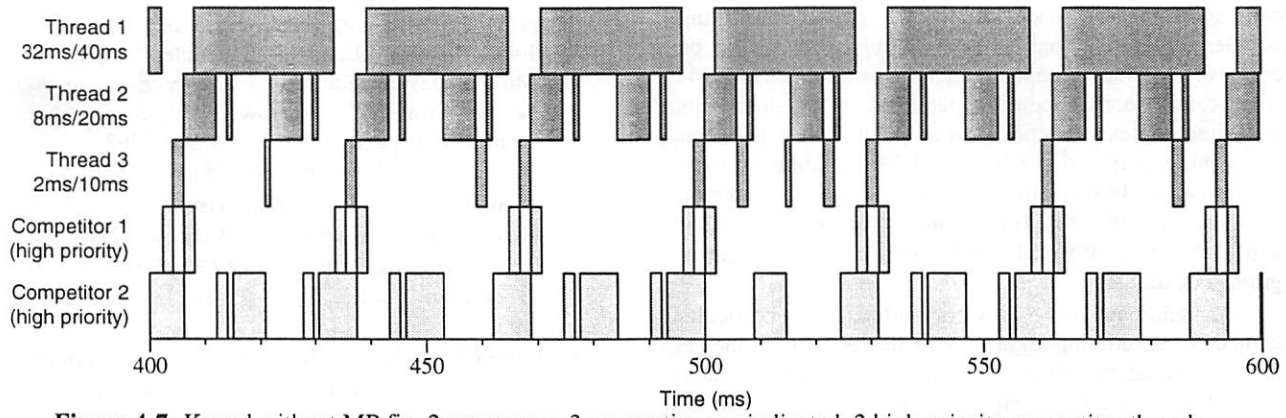


Figure 4-7: Kernel without MP fix, 2 processors, 3 reservations as indicated, 2 high-priority competitor threads

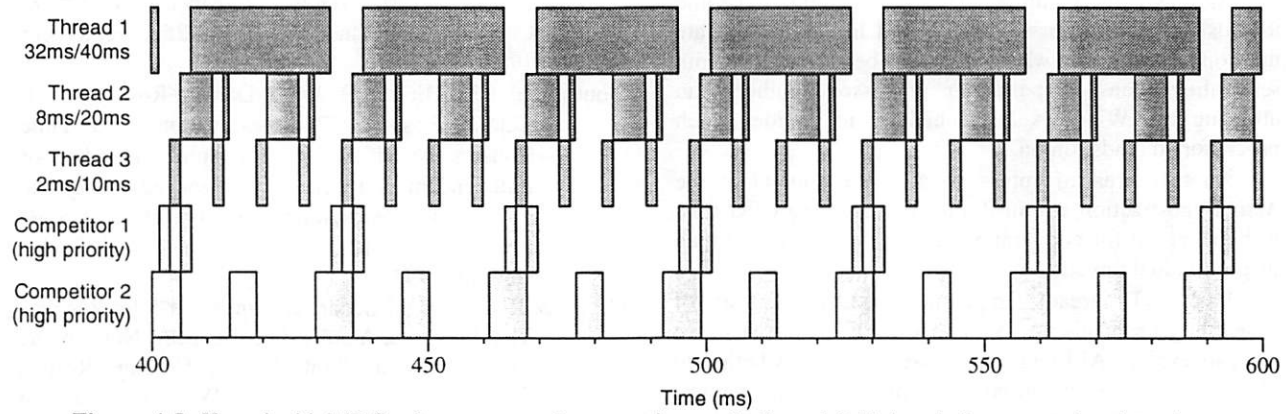


Figure 4-8: Kernel with MP fix, 2 processors, 3 reservations as indicated, 2 high-priority competitor threads

candidates for preemption, the RT threads always ran during their reserved time slices.

5. Methodology

While developing Rialto/NT, we wrote a small (1500 lines of C) event-driven simulator that simulates the parts of the kernel environment relevant to scheduling. The scheduler can be compiled either into the kernel or the simulator. Being able to easily switch between the two has been essential to the development process for a number of reasons: the lack of real concurrency in the simulation ensures that any bugs we find using it are functional bugs rather than races, and the deterministic nature of the simulator allows us to keep replaying a troublesome scenario until we get it right. The debug cycle is much shorter since it does not include a reboot, and in the Visual C++ environment we can use sophisticated tools like a graphical debugger, Purify, and BoundsChecker. We can also turn on or off complications such as late DPCs at will in order to debug the code that handles these conditions.

6. Related Work

The goal of this work is to investigate the feasibility of bringing benefits of predictable Rialto-style scheduling [Jones et al. 97] to Windows NT applications. This having been said, we want to contrast our approach with some alternative paths that could be taken.

One possibility would be to use Windows NT as is for time-sensitive applications. This can work acceptably when only one application is run at once since scheduling contention may not occur. Likewise, multiple time-sensitive applications can coexist provided sufficient resources exist to run all of them and they happen to not interfere with one another's execution. Unfortunately, interference appears to be all too common, even between a single time-sensitive application and other active tasks.

Another possible approach is to augment Windows NT with a separate add-on real-time kernel. For instance, VenturCom sells a real-time kernel called RTX [Carpenter et al. 97] that replaces the HAL beneath Windows NT, allowing applications using its new system services to obtain predictable real-time scheduling.

In contrast, by building predictable scheduling facilities into Windows NT itself, it is our goal to allow applications to predictably obtain guaranteed amounts of CPU time, while still using normal Win32 APIs.

Rialto/NT adds new scheduling mechanisms to the Windows NT kernel, while using the kernel's native priority scheduler to actually dispatch threads. In contrast, [Lin at al. 98] reports on a system that likewise uses the

priority scheduler to dispatch soft real-time threads but does so from user space and using different scheduling policies. While they have shown that this approach can be effective in some contexts, their redispatch mechanism is significantly more expensive, requiring six system calls and three context switches [Lin at al. 98, p. 153]. Their dispatching overhead is 640µs or 3.2% for 20ms periods, as opposed to 18.6µs (up from the kernel's native 10.6µs) for ours or 1.9% for 1ms periods. Nonetheless, their approach can work well for tasks with sufficiently coarse-grained deadlines.

A significant body of work pertaining to particular choices of scheduling algorithms is discussed in [Jones et al. 97]. Readers interested in this aspect of the related work should review its treatment there.

7. Further Research

Although our modified kernel can reliably schedule threads on multiprocessors, we would like to investigate the conditions under which it would be desirable to pin scheduling plans to particular processors, rather than allowing the Windows NT scheduler to decide which processors threads run on.

Another area of future work is to implement the *Activity* abstraction within Rialto/NT, allowing CPU time to be reserved for cooperating sets of threads, rather than just individual threads.

Rialto/NT already implements a flexible set of scheduling mechanisms. Now that those are in place, we need to explore API and policy issues such as whether to allow forms of reservations and constraints that request that they occur on a particular CPU. Likewise, other possible higher-level requests such as "please schedule me on the same (or a different) CPU as that reservation" could be investigated. Co-scheduling within this framework is another obvious area of possible research.

Finally, and most importantly, we plan to use Rialto/NT scheduling in an attempt to improve the usefulness of a number of real applications.

8. Conclusions

This research demonstrates that the *Precomputed Scheduling Plan* data structures originally developed for the Rialto operating system to implement *CPU Reservations* and *Time Constraints* can be effectively extended to schedule shared-memory multiprocessors.

We have presented encouraging early results from a reimplementation of these abstractions within a research version of Windows NT called Rialto/NT. While not yet as mature as the Rialto implementation, these results have already demonstrated the effectiveness and practicality of implementing CPU Reservations and Time Constraints on a multiprocessor operating system and within Windows NT in particular.

Acknowledgments

The authors wish to thank Patricia Jones for her editorial assistance in the preparation of this manuscript.

References

[Candea & Jones 98] George M. Candea and Michael B. Jones. Vassal: Loadable Scheduler Support for Multi-Policy Scheduling. In *Proceedings of the Second USENIX Windows NT Symposium*, Seattle, WA, pages 157-166, August 1998.

[Carpenter et al. 97] Bill Carpenter, Mark Roman, Nick Vasilatos, and Myron Zimmerman. The RTX Real-Time Subsystem for Windows NT. In *Proceedings of the USENIX Windows NT Workshop*, Seattle, WA, pages 33-37, August 1997.

[Jones et al. 96] Michael B. Jones, Joseph S. Barrera III, Alessandro Forin, Paul J. Leach, Daniela Roşu, Marcel-Cătălin Roşu. An Overview of the Rialto Real-Time Architecture. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, Connemara, Ireland, pages 249-256, September 1996.

[Jones et al. 97] Michael B. Jones, Daniela Roşu, Marcel-Cătălin Roşu. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, St-Malo, France, pages 198-211, October 1997.

[Jones & Regehr 99] Michael B. Jones and John Regehr. The Problems You're Having May Not Be the Problems You Think You're Having: Results from a Latency Study of Windows NT. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems (HotOS-VII)*, Rio Rico, Arizona, March 1999.

[Lin at al. 98] Chih-han Lin, Hao-hua Chu, and Klara Nahrstedt. A Soft Real-time Scheduling Server on the Windows NT. In *Proceedings of the Second USENIX Windows NT Symposium*, Seattle, WA, pages 157-166, August 1998.

[Microsoft 99] Windows NT 4.0 DDK Documentation, *MSDN Library*. Microsoft, April 1999.

[Solomon 98] David A. Solomon. *Inside Windows NT, Second Edition*. Microsoft Press, 1998.

Hard Real-Time With RTX on Windows NT

Mike Cherepov, Chris Jones
{cher, clj}@vci.com
VenturCom, Inc.
Cambridge, MA
www.vci.com

Abstract

For a variety of reasons, Microsoft Windows NT is increasingly being considered as a platform for deployment of real-time systems. In order to meet the stringent latency requirements of hard real-time systems, it is necessary to augment the capabilities of Windows NT. We examine VenturCom's RTX, which provides a real-time subsystem running on Windows NT. It implements deterministic scheduling of real-time threads, inter-process communication mechanisms between the real-time environment and the native NT environment, and other extensions to Windows NT which are often found in specialized real-time operating systems. We discuss how the components of RTX provide these features, explore current results and experiences, and point out possible future directions for enhancement.

1 Introduction

Microsoft Windows NT's popularity and market share have been growing. The reasons for this are varied, including:

- The increasing power and declining price of Windows NT platforms.
- The many applications available on the platform.
- The variety of development tools available on the platform.
- The richness of the Win32 Application Programming Interface (API).
- The large number of developers, support personnel, and end users who are familiar with the system.

Because of the added complexity and cost of maintaining a heterogeneous computing environment, more companies are striving to use Windows NT as their Operating System (OS) at all levels of the industrial hierarchy. Its use as a network server system or as a desktop system is easy to understand, since these are the very environments for which Windows NT was designed. However, there is also an impetus to use it in other environments, such as the factory floor. A common characteristic of these environments is that they often require hard real-time system behavior.

Can Windows NT fulfill this need? The answer is, not as delivered. However, with additional software, it is possible to realize hard real-time performance on Windows NT. The remainder of this paper justifies these statements and describes VenturCom's RTX environment including RTSS, a Real-Time SubSystem, for Windows NT running on the PC architecture (i.e., Intel x86 and compatible systems).

An earlier paper [Carpenter 97] discussed this effort during its development. This paper offers a closer look at the actual implementation as delivered, including performance numbers and enhancements made in the meantime, as well as a look at future areas for development.

2 Windows NT and the Real-time World

2.1 What it means for a system to be real-time

A real-time system is one in which the correct operation of the system depends not only on the results that are delivered, but **when** they are delivered. It is important to note that "real-time" does not necessarily mean "fast"; rather it refers to how deterministic the response time characteristics of the system are. That is, the important measure is not average response time but worst-case response time. Real-time systems are sometimes further classified as *hard* or *soft* real-time systems. A hard real-time system is one in which the response time determinism requirement is absolute; for a soft real-time system, some small deviations are tolerated. A fundamentalist viewpoint would consider "soft real-time" to be an oxymoron, and for the remainder of this paper, when we say "real-time" we mean hard real-time.

An example of a real-time system is a system controlling a piece of capping machinery over a conveyer belt transporting bottles to be capped. It is not enough for

the system to correctly position the cap dispenser; it must do so when a bottle is in position to be capped. All the accuracy in positioning is worthless if the dispenser arrives in position after a bottle has passed.

In addition to this determinism, there are a number of other requirements that real-time systems have typically come to provide:

- A multithreaded, preemptive scheduler with a large number (typically 64-256) of thread priorities.
- Predictable thread synchronization mechanisms.
- A system of priority inheritance.
- Fast clocks and timers.

2.2 Why is stock Windows NT unsuitable as a real-time system?

Microsoft Windows NT has been designed as a general-purpose operating system, suitable for use both as an interactive system on the desktop and as a server system on a network [Solomon 98]. The shortcomings of NT in real-time applications have been thoroughly researched [Ramamritham 98] [Timmerman & Monfret 96]:

- Too few thread priorities.
- Opaque and non-deterministic scheduling decisions.
- Priority inversion, particularly in interrupt processing.

The logic of RTX design is dictated by several factors. The stock NT operating system is a mass-market product, not readily tweaked for niche applications like real-time. While Microsoft-sponsored research into real-time NT has produced some interesting results [Sommer 96], especially for cases when applications advertise their resource requirements in advance [Sommer 97][Sommer & Potter 96], it is doubtful that this operating system, aiming for a very broad market, should absorb the overhead and complexity of real-time functionality [Microsoft 95]. This factor suggests that the proper way to add real-time to NT is via an extension, or a plug-in to the generic product [Jones 98].

2.3 Why extend Windows NT to a real-time system?

At the same time, NT offers a very rich and sophisticated device driver model. That, and the customizable Hardware Abstraction Layer (HAL), offer a developer great flexibility and control over system behavior, and scope for creativity in tackling technical challenges. Thus, real-time functionality can be implemented "by the book" following the Microsoft NT Device Driver Kit (DDK) and HAL models [Baker 97].

Finally, for a writer of NT kernel extensions who is not employed by Microsoft, the NT kernel is akin to silicon, as its interfaces and behavior are fixed. Rather than lament the fact, one can make virtue of necessity and produce a compact design for the extension, easily portable between different versions of NT, and, indeed, between NT and other operating systems such Windows CE or Unix. Below we illustrate how RTX lived up to these portability goals.

3 So, You Want to Write A Hard Real-Time NT Environment?

3.1 Why extend Windows NT to a real-time system?

Given that many of the shortcomings of NT just mentioned are due to its thread model and thread scheduler, it is logical that the extension have its own thread model with its own scheduler. Likewise, the NT syn-

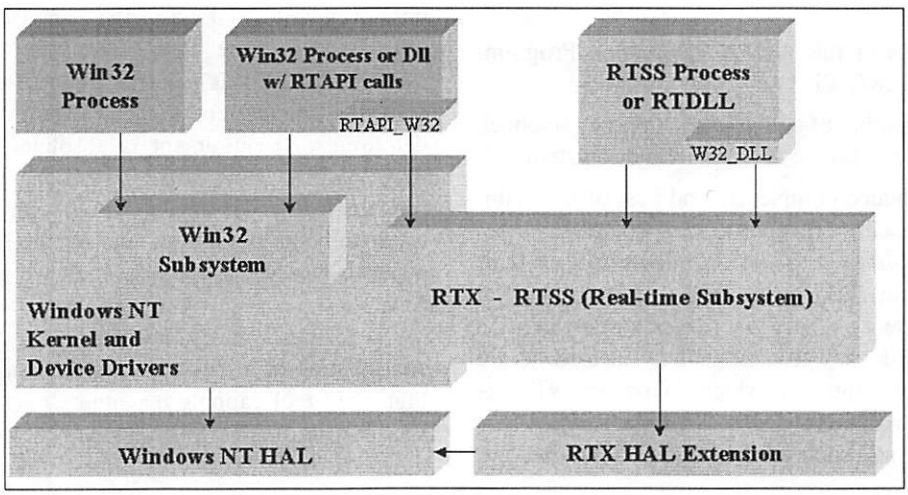


Figure 1. RTX Architecture

chronization objects such as events, semaphores, and mutexes lack the necessary real-time semantics (in particular, they neither ready threads waiting on an object in priority order nor prevent priority inversion). For these reasons, the extension should implement its own synchronization objects [Bollella 95].

If, following the logic of the NT environment, you have decided to implement a hard real-time subsystem for NT, your real-time environment should be able to:

- Preempt NT anywhere, at least outside critical NT interrupt-processing code.
- Defer NT interrupts and faults while running real-time tasks.
- Process real-time interrupts while running real-time tasks.

The notion of preempting high-level IRQ activity of NT and its drivers for unbounded periods of real-time activity may strike one as dangerous. Yet, such events are commonplace, and NT is designed to handle them: high-IRQL events intrude on lower-IRQL ones, bus-mastering by DMA peripherals may defer even the highest-level interrupt processing, and PCI devices may stall CPU accesses to the I/O space (see RTX and Interrupt Latency). Thus, from the NT point of view, RTSS activity that steals its cycles is equivalent to taking and coming back from an interrupt. Such an event is well handled by NT, regardless of its duration.

Functional needs of the real-time subsystem would include IPC with the Win32 subsystem, access to the NT kernel functionality (interrupt management, port I/O, shutdown/crash handlers), and – importantly – compatibility with Win32, at least at the source code level.

3.2 RTX Structure

RTX is implemented as a collection of libraries (both static and dynamic), a real-time subsystem (RTSS) realized as an NT kernel device driver, and an extended HAL (see Figure 1) [Carpenter 97]. The subsystem implements the real-time objects and scheduler previously mentioned. The libraries provide access to the subsystem via a real-time API (RTX API). RTX API provides access to these objects. Note that the RTX API is callable from the standard Win32 environment as well as from within RTSS. While using RTX API from Win32 does not provide the determinism available within RTSS, it does allow much of the application development to be done in the friendlier Win32 programming environment rather than that provided by the DDK [Anschuetz 98]. All that is necessary to convert a Win32 program to an RTSS program is to relink with a different set of libraries.

RTX applications – just as RTX itself – are implemented on top of loadable NT drivers, although they lack the I/O Manager-related hooks. It is a natural fit: Windows NT drivers are process-like from the NT Service Manager's point of view, controllable by users, and get loaded into the kernel address space.

4 RTX in Depth

4.1 The Real-time HAL

The HAL is the one piece of the Windows NT system whose source is available for modification and extension. RTX has modified the HAL for three purposes:

1) To add interrupt isolation between NT and RTSS threads.

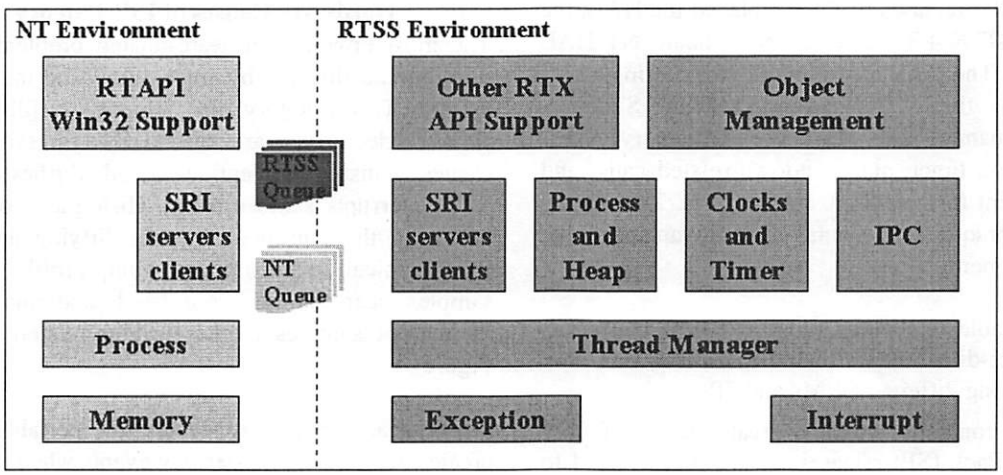


Figure 2. RTSS Detailed Architecture

2) To implement high-speed clocks and timers.

3) To implement shutdown handlers.

Interrupt isolation means that it is impossible for an NT thread or an NT-managed device to interrupt RTSS. It is also impossible for an NT thread to mask an RTSS-managed device. The HAL ensures that these conditions are met by controlling the processor's interrupt mask. When running an RTSS thread, all NT-controlled interrupts are masked out. When an NT thread calls to set the interrupt mask, the HAL, which is the software that actually manipulates the mask, ensures that no RTSS-controlled interrupt is masked out.

NT provides clocks and timers with a maximum resolution (i.e., smallest granularity) of 1 msec. The RT-HAL extends this to 1 µsec. It also provides access to the second clock on the PIC.

4.1.1 Protection from NT Crashes

In addition to interrupt management and fast-timer services, the real-time HAL also provides NT shutdown management. An RTSS application can attach an NT shutdown handler for cases where NT performs an orderly shutdown, or crashes, producing a so-called Blue (Stop) screen. An orderly shutdown allows RTSS to continue unimpaired and resumes when all RTSS shutdown handlers return. In a blue-screen shutdown, RTSS shutdown handlers run with certain limitations, unable to call NT services (e.g., for memory allocation) or to handle faults. In practice, it means that a shutdown handler should clean up and reset any hardware state, possibly alert an operator, or switch to a hot spare when the system stops, due to either a normal shutdown or a crash.

4.2 Extending the HAL

Whereas earlier releases of RTX replaced the HAL, the latest one – RTX 4.3 – extends the standard NT HAL dynamically. The HAL extension driver, starting at OS initialization time (SERVICE_SYSTEM_START), performs dynamic HAL detection in memory, intercepts interrupt, timer, and shutdown-related calls, and re-directs them to their RTX counterparts. This binary hooking technique has a number of advantages over HAL replacement:

- RTX handles a broader range of OEM platforms, as call re-direction is limited to calls which vary little among different OEMs and SPs.
- RTX is compatible with a greater range of NT Service Pack (SP) releases, as we do not need to merge RTX changes with the latest SP's HAL sources.
- Installation becomes more robust, as the on-disk copy of the HAL is untouched, hence RTX is unaffected by SP upgrades.
- Upgrades to newer versions of NT become easier, if not effort-free.

The benefits of the HAL extension were demonstrated when RTX has installed and run successfully on Windows 2000 (neè NT 5.0) Beta 2. This required no development effort on Win2000, although effort certainly will be needed to improve performance to NT4.0 levels.

4.3 RTX and Interrupt Latency

4.3.1 Software Causes of Latency

A switch from NT to RTSS happens on an interrupt, either from the RTX high-speed clock, or from another device generating RTX interrupts. Therefore, achieving RTX ISR determinism requires reducing NT interrupt latency. Let us examine the sources of this latency.

The most significant is IRQ masking by the NT kernel and drivers, routinely done for periods up to several milliseconds via NT *KeRaise/LowerIrql* calls. The NT kernel, HAL, and certain special drivers also perform processor-level masking of all interrupts via x86 STI/CLI instructions, for periods of up to 50 µsec.

NT and RTX interrupt processing, naturally, masks interrupts, thereby adding to ISR latency. Although NT relies very heavily on interrupts in many situations (e.g., raising software exceptions, or unwinding a thread's stack), the NT interrupt sequence is reasonably compact in its contribution to the worst-case ISR latency.

4.3.2 Hardware Causes of ISR Latency

The most obvious hardware-related problem is cache dirtying and flushing by applications and the operating system. This category also includes re-filling of the TLBs. Video drivers are particularly aggressive users of caches, causing contention-related flushes when an RTX interrupts starts running. Histograms of ISR behavior in the presence of cache-dirtying applications would typically have a double-hump profile, with most samples near the best-case band, and another large number of samples in the flushing-related band (see Figure 3).

Power management, especially on portable devices, creates occasional long-latency events when the CPU is put in a low-power-consumption state after a period of

inactivity. Such problems are usually quite easy to detect. A typical system can disable those features via BIOS setup.

Bus mastering events can cause long-latency CPU stalls. Such cases include high-performance DMA SCSI devices, causing CPU stalls for periods of many microseconds, or video cards that insert wait cycles on the bus in response to a CPU access. Sometimes the behavior of such peripherals can be controlled from the driver, trading off throughput for lower latency.

While no operating system can protect an application against such hardware factors, RTX offers a panoply of tools to diagnose platform-related latencies, and identify the misbehaving peripherals. Being mindful of such factors and using RTX tools to qualify one's development platform are essential for a system's overall performance.

4.4 RTX Interrupt Latency Reduction Techniques

RTSS entirely eliminates latencies from IRQL masking by NT and NT drivers. The RT HAL performs interrupt isolation, re-programming the PIC when switching between NT and RTSS. The result is that RTX interrupts can always interrupt NT, while RTX masks all NT interrupts while RTSS is running.

Processor-level interrupt masking, on the other hand, can not be defeated, other than through the perilous use of x86 NMIs (non-maskable interrupts). RTX adopts a static solution hooking gratuitous cases of interrupt preemption (e.g., page-zeroing operations) to use IRQ locks instead. The RTX Dynamic Hook driver scans the NT kernel for signatures of such operations, hooking them to use spin locks (or IRQ-based synchronization on a uniprocessor) instead.

These techniques provide worst-case interrupt latencies of under 50 microseconds on a typical 200MHz PC platform.

4.5 RTX Objects

The RTSS Environment has a fast streamlined object manager (see Figure 2). The objects it supports satisfy the following criteria: 1) Usefulness for real-time programming, and 2) Compatibility with Win32. The IPC objects are also available to Win32 applications and device drivers, allowing programmers to harness the full power of NT. The IPC set includes mutexes, events, semaphores, and shared memory objects.

The RTSS object manager uses the Windows NT nonpaged memory pool for its storage requirements. There are advantages and disadvantages to this approach. Using kernel-provided mechanisms decreases RTX's development time and resource consumption. Object allocation, however, is non-deterministic.

4.6 RTSS Scheduler

The RTSS scheduler implements a priority based preemptive policy with priority promotion to prevent priority inversion. The RTSS environment provides for 128 priority levels, numbered from 0 to 127, with 0 the lowest priority. The RTSS scheduler will always run the highest priority thread that is ready to run (in the case of multiple ready threads with the same priority, the thread which has been ready the longest will run first). An RTSS thread will run until a higher priority ready thread preempts it or until it voluntarily relinquishes the processor by waiting (there is no time-slicing among ready threads at the same priority).

The scheduler has been coded with the requirements of real-time processing in mind. Most importantly, its operation is low latency, and is unaffected by the number of threads it is managing. Each priority has its own ready queue, maintained as a doubly linked list. This allows the execution time of insertion (at the end of the list) and removal (from anywhere in the list) to be independent of the number of threads on the list. A bit array keeps track of which lists are non-empty, and manipulating this bit array is done by high-speed assembly-coded routines.

While an RTSS thread is running, all NT-managed interrupts, as well as any interrupts managed by threads of a lower priority than the current thread, are masked out. Conversely, all interrupts managed by higher priority threads are unmasked, allowing for a higher-priority thread to preempt the current thread. In addition to these device interrupts, other mechanisms that can cause the currently running thread to be preempted are the expiration of a timer that causes a higher priority thread to become ready, or the signaling of a synchronization object (by the currently running thread) for which a higher priority thread is waiting.

In order to deal with priority inversion, RTSS implements the classic solution [Nakajima 93] [Sha 90], *priority promotion,* to prevent this situation. For the duration of the time that a low priority thread owns an object for which a high priority thread is waiting, its effective priority is promoted to that of the high priority thread.

4.7 Service Request Interrupt (SRI)

An important architectural feature of RTX is its lockless interrupt-driven interface between NT and RTSS. This clean architectural separation has enabled ports of RTSS to various environments (e.g., multiprocessor RTX product, and RTSS demo for Windows CE2.0), while ensuring a fast and robust implementation. The NT side of the RTX driver and the RTSS environment communicate by inserting commands into one of the two buffer queues (one in each direction) and initiating a Service Request Interrupt (SRI) to request service by the other side. A server thread executes a request and a reply message is posted in the other buffer. A typical NT-to-RTSS request is an IPC operation like WaitForSingleObject or a Release operation on a RTSS object. A typical RTSS-to-NT operation is a memory allocation or a file I/O request. The SRI design favors lower response time over throughput, responding to an RTSS request as soon as possible.

4.8 Win32-RTSS IPC

Inter-environment IPC is a key feature of RTX, allowing tightly integrated applications where hard real-time processes run in the more resource-intensive RTSS environment, and the rest of the application runs in the Win32 subsystem. This section describes the IPC design.

4.8.1 RTSS Proxy Model

IPC, as the rest of the NT-RTSS communication, uses the SRI channel. Given that the SRI channel prevents NT threads from queuing directly for RTX objects, RTX uses *proxy* processes and threads to support blocking IPC from Win32. When a Win32 thread accesses an RTX object, RTSS uses a proxy thread on its behalf. This model is clean and economical, its advantages being:

- No state-keeping on the NT side for blocking IPC requests.
- No special-casing in RTSS for external Win32 wait requests.
- Handle and object cleanup for Win32 process and thread termination is handled automatically by RTSS proxy process/thread cleanup.

Although proxies involve some memory and CPU overhead, the clean design and quick implementation were worth the tradeoff.

4.8.2 Taming the NT I/O Manager

Preserving seamless Win32 semantics and achieving good performance for cross-environment IPC presents several challenges.

The NT 4.0 DDK provides no exposed interfaces for driver thread notification in case a Win32 thread using that driver terminates. Yet, Win32 mutex semantics, require such a mechanism. An RTX mutex acquired, but not released, by a thread at the time of the thread's termination, must be marked as "abandoned", indicating that the shared data it protects may be inconsistent. To implement thread-termination cleanup, RTX takes advantage of I/O Manager's IRP (I/O Request Packet) cleanup: each thread attached to the RTX Win32 Dynamically Linked Library (DLL) sends a "death IRP" to the RTX driver. When the thread terminates, NT calls this IRP's cancel routine, notifying the RTX driver and, thereby, the RTSS object layer. The I/O Manager presents a powerful, yet often challenging environment, as its event delivery is asynchronous. E.g., calls to the MJ_CLEANUP driver dispatch routine and the cancel routine call can come in any order, requiring careful synchronization for the RTX driver's per-thread and per-process structures.

Performance of the Win32-RTSS IPC presented another concern. In an early implementation, the total latency of an uncontested RtWaitForSingleObject call from Win32 averaged 130μsec. Analysis has shown that about 40μsec of the total (over 30%) was spent in the NT I/O Manager. Therefore, RTX4.2 has undergone a redesign of the IPC code, using direct signaling and shared memory between the RTX driver and the Win32 IPC client [Tomlinson 97]. RTX user and kernel threads share synchronization objects, signaling each other directly, thus shrinking the overhead and the latency of the NT I/O manager by a factor of four.

Note that operations on RTX synchronization objects locked by Win32 applications become nondeterministic [Carpenter 97]: as any RTSS thread can preempt an NT thread holding such a lock, causing an apparently unbounded case of priority inversion. This, however, is a matter of application design: locking an object shared from Win32 should be left to a noncritical RTSS thread. Furthermore, this issue is ameliorated when RTSS and NT run on different dedicated processors, in a multiprocessor RTX system.

4.9 Fast Timer Support

On all PC platforms real-time HAL provides clock resolution of 1μsec or better, and timer period of

100μsec or better. When RTSS is not used, there are no timing differences between a real-time HAL and a regular HAL systems.

4.10 Dynamically Linked Libraries

No tour of Win32 would be complete without a mention of DLLs. RTSS supports Win32 DLL API (LoadLibrary, GetProcAddress). Currently, all the static and global data in an RTSS DLL is shared between all RTSS processes attached to that DLL.

4.11 Structured Exception Handling in RTSS

Structured Exception Handling (SEH) is a relatively little known but rather important feature of Win32 and NT kernel environments. Its pedigree goes back to OS/2 and OSF Unix implementation. SEH provides exception handling via *try/except* and *try/finally* constructs of the Microsoft C implementation. C++ exception handling is layered on top of SEH, as are libc *signal/raise* calls, making SEH a necessity for any Win32-compliant environment. The salient features of this model are:

- Compiler-specific exception handlers.
- Operating system-specific stack unwinder and exception dispatcher routines.
- User-supplied exception filters.
- Two-stage exception handling algorithm which first invokes the OS-specific dispatcher routine to scan the thread's stack backwards calling filters in search of a suitable handler, then the OS-specific unwinder to roll back the stack, if necessary.
- Last-chance default and user-supplied exception routines.
- A special mechanism for nested exceptions and "collided" unwinds.

The RTSS SEH implementation maintains compatibility with Microsoft structures, handler-calling conventions, SEH API behavior, etc. In addition, it is engineered for real-time, to minimize processor-level interrupt masking and disruptions to RTSS threads:

- Win32 generates a software interrupt when raising a software exception via the Win32 RaiseException API; RTSS calls a user-mode exception dispatcher.
- Win32 SEH uses a special interrupt when it sets a new user context after unwinding the stack; RTSS restores context in user mode.
- NT may "edit" (move) an exception trap frame with interrupts disabled; RTSS does this in user mode.

For hardware exceptions, the RTSS algorithm doctors the trap frame to call the RTSS exception dispatcher; then "returns" from the ISR to the dispatcher, and handles the exception. Thus, a software exception involves no ISR latency penalty for other threads; a hardware exception only adds the worst-case penalty of a single interrupt.

5 Performance

Table 1 presents selected performance numbers for a recent release of RTX.

Table 1. Typical latencies for a 266MHz Pentium II

Operation	*Latency (μsec)*
Interrupt thread: avg., max	9, 38
Timer interrupt: avg., max	10, 43
Context switch (yield): avg.	3.5
Context switch (semaphore): avg.	5.5
Thread priority change: avg.	3
Thread yield: avg.	3
Acquire semaphore, uncontested: avg.	1.5
Acquire semaphore: avg.	5
Win32-to-RTSS semaphore call: avg.	50

Figure 3 is a typical look from the RTSS interrupt latency measurement utility in the presence of a typical NT workload: disk searches, video updates, network activity, screen saver, etc. The lower and narrower sections of the chart's bars represent activity during the last second only.

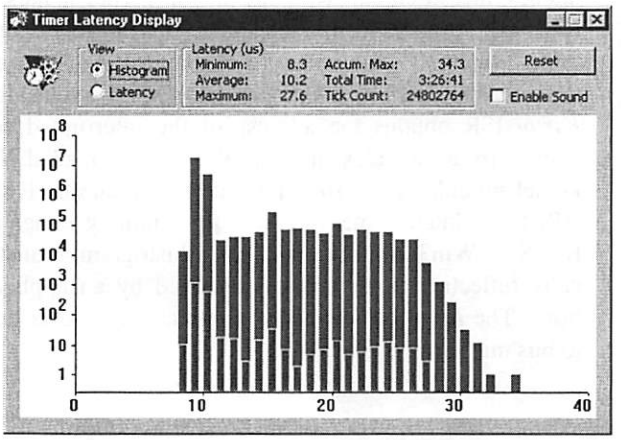


Figure 3. RTSS timer interrupt latency histogram for a typical workload. X axis – μsec, Y axis – number of samples.

Figure 4 is the Win32 counterpart for the same kind of workload on the same machine – Gateway GP5-166 running NT4.0 Workstation SP4. An astute reader may point out that the first histogram describes Win32 threads, whereas the second – kernel driver threads. However, the difference between the two amounts to the several microseconds of kernel-user transition overhead for the Win32 threads, so the worst-case latencies of NT driver and Win32 threads differ very little under the same workload.

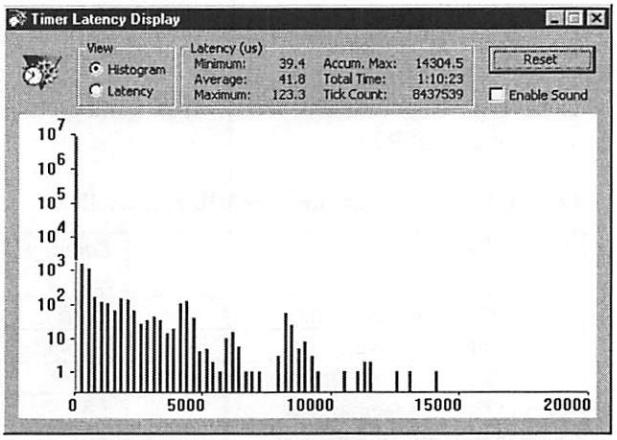


Figure 4. Win32 timer interrupt latency histogram for a typical workload. X axis – μsec, Y axis – number of samples.

Copious performance results are available from independent evaluations of RTX and other real-time NT extensions [OMAC 98] [Real-Time 98] [Timmerman 98].

5.1 Performance Tools

RTX performance tools enable developers to qualify and tune performance on their platforms. *Ksrtm* is a driver and a Win32 utility for HAL-level timer interrupt latency measurement. Running in the kernel makes it relatively insensitive to cache jitter. *Ksrtm* also determines which NT component or device driver is causing the greatest latency event: after such an event, the *Ksrtm* ISR obtains the address of the interrupted sequence from the stack, then resolves it to a loaded NT kernel module. The *Srtm* application is a simple RTX API timer latency measurement tool running either in RTSS or Win32, which produces a histogram, realistically reflecting timer latency observed by an application. The *Lpt* tool determines long-latency events due to bus mastering events.

6 Future Directions

6.1 Multiprocessing

The initial releases of RTX ran on single processor systems. The most recent release runs on multiprocessor systems that conform to the Intel Multiprocessor Specification Rev 1.4.

This specification provides for interrupts to be controlled by an Advanced Programmable Interrupt Controller (APIC), suitable for a multiprocessor system. Through the APIC, different interrupts can be steered to different sets of processors. RTSS dedicates one processor of the system to running RTSS threads, while the remaining processors run NT threads. This dramatically lessens the latency of real-time threads (from 50 μsec to less than 13 μsec,) while preventing the processor starvation of NT threads, possible on a single processor system

6.2 Ways to Grow

RTSS can serve as a basis for other real-time environments, like Newmonics' real-time RTSS-based Java-compatible virtual machine [Nilsen 98] [Nilsen & Lee 98]. It also can be linked with other NT subsystems such as Interix by Softway [Walli 97]. Other possibilities include a COM interface to RTSS objects accessed from Win32 or, indeed, making COM available within the RTSS environment.

7 Conclusion

VenturCom's RTX has shown that, with a selected collection of extensions, it is possible to augment Windows NT to provide the features of a real-time operating system at the same time it continues to be used as a general purpose platform. The resulting system meets the constraints of determinism which are a necessary part of the real-time world, while providing an environment more familiar to a wide body of users.

Availability

RTX is available from www.vci.com.

References

[Anschuetz 98] E. Anschuetz, M. Biddle, S. Giambarberee, B. Riner, J. Dube, *Real Time Flight Simulators Under NT*, Proceedings of the 1998 Interservice/Industry Training, Simulation and Education Conference (I/ITSEC 12 1998).

[Baker 97] Art Baker, *The Windows NT Device Driver Book*, Prentice Hall, 1997.

[Bollella 95] G. Bollella and K. Jeffay, *Support For Real-Time Computing Within General Purpose Operating Systems: Supporting co-resident operating systems*, Proc. IEEE Real-Time Technology and Applications Symposium Chicago, IL, May 1995.

[Carpenter 97] Bill Carpenter, Mark Roman, Nick Vasilatos, Myron Zimmerman, *The RTX Real-Time Subsystem for Windows NT*, Usenix Windows NT Symposium, 1997.

[Jones 98] C. Jones, M. Cherepov, *Windows-based systems and the Win32 API*, Real-Time Magazine 98/3.

[Microsoft 95] Microsoft, *Real-Time Systems and Microsoft Windows NT;* http://www.msdn.microsoft.com.

[Nakajima 93] T.Nakajima, T.Kitayama, H.Arakawa, H.Tokuda , *Integrated Management of Priority Inversion in Real-Time Mach*, IEEE Real-Time Systems Symposium, December 1993.

[Nilsen 98] Kelvin Nilsen, Simanta Mitra, Sairam Sankaranarayananan, Venkatesh Thanuvan, *Asynchronous Java Exception Handling in a Real-Time Context*, Workshop on Programming Languages for Real-Time Industrial Applications '98, Madrid, Spain, December 1, 1998.

[Nilsen & Lee 98] Kelvin Nilsen, Steve Lee, *Perk(TM) Real-Time API,* July 1998, Newmonics, Inc. www.newmonics.com/WebRoot/perc.info/perc.api.pdf.

[OMAC 98] Open Modular Architecture Controls (OMAC) Users Group, *Hard Real-time Extensions of Windows NT© Evaluation Report Test Plan and Phase 1 & 2*; www.arcweb.com/omac/Docs&NRs/ntrtrpt2.pdf.

[Ramamritham 98] Krithi Ramamritham, Chia Shen, Oscar Gonzalez, Shubo Sen, Shreedhar B Shirgurkar, *Using Windows NT for Real-Time Applications: Experimental Observations and Recommendations* Proceedings of IEEE RTAS'98 (the IEEE Real-Time Technology and Applications Symposium), June 3-5, 1998. Denver, Colorado.

[Real-Time 98] Real-Time Magazine, *Windows NT RT Extensions - Evaluation Reports;* www.realtime-info.be/encyc/market/rtos/eval_roadmap.htm.

[Sha 90] L. Sha, R. Rajkumar, and J. P. Lehoczky, *Priority Inheritance Protocols: An Approach to Real-Time Synchronization*, IEEE Transactions on Computers, 39 (9):1175-1185, Sept. 1990.

[Solomon 98] David A. Solomon, *Inside Windows NT, Second Edition*, Microsoft Press, 1988.

[Sommer 96] Sommer, S., *Removing Priority Inversion from an Operating System,* Proceedings of the Nineteenth Australasian Computer Science Conference (ACSC'96), Melbourne, Australia, January 31 - February 2, 1996.

[Sommer & Potter 96] S. Sommer and J Potter, *An Overview of the Real-Time Dreams Extensions* Proceedings of the Third Australasian Conference on Parallel and Real-Time Systems, September 1996.

[Sommer 97] Sommer, S., *Dreams in a Nutshell*, Proceedings of the USENIX Windows NT Workshop, Seattle, Washington, August 11-13, 1997.

[Timmerman & Monfret 96] M. Timmerman and J. Monfret, *Windows NT as Real-Time OS?* Real-Time Magazine; http://www.realtime-info.be.

[Timmerman 98] Martin Timmerman, Bart Van Beneden, Laurent Uhres, *Windows NT Real-Time Extensions: better or worse?* Real-Time Magazine - 98/3.

[Tomlinson 97] Paula Tomlinson, *Understanding NT: Signaling Apps from Drivers,* Windows Developer's Journal, March 1997.

[Walli 97] Stephen R. Walli, *OPENNT: UNIX Application Portability to Windows NT via an Alternative Environment Subsystem,* Proceedings of the USENIX Windows NT Workshop, Seattle, Washington, August 11-13, 1997.

Higher-Order Concurrent Win32 Programming

Riccardo Pucella
Bell Laboratories
Lucent Technologies
riccardo@research.bell-labs.com

Abstract

We present a concurrent framework for Win32 programming based on Concurrent ML, a concurrent language with higher-order functions, static typing, lightweight threads and synchronous communication channels. The key points of the framework are the move from an event loop model to a threaded model for the processing of window messages, and the decoupling of controls notifications from the system messages. This last point allows us to derive a general way of writing controls that leads to easy composition, and can accommodate ActiveX Controls in a transparent way.

1 Introduction

Programming user interfaces on the Windows operating system at the level of the Win32 API, its lowest level, is hard under the best conditions and maddening the rest of the time. At least three points underlie the difficulty:

- the API is large (more than a thousand functions) and growing;
- the system is input-driven, making it difficult to perform intensive computations and remain interactive;
- the system is centered around an event loop, yielding control-flow that is hard to understand.

The first problem is hard to circumvent, considering that the API is large because it covers a lot of functionality required by applications. However, a better structuring can help reduce the burden of using the API, as demonstrated by the popularity of C++ class frameworks such as *Microsoft Foundation Classes* (MFC) or Borland's *Object Windows Library* (OWL). The remaining two problems are in fact related, consequences of Win32 being a message-passing API based on callback procedures [12].

It has been recognized that to ensure interactivity of applications in the presence of computation-intensive code, multiple threads should be used [2, 5]. It turns out that by using a truly concurrent approach rather than the system-level threads available from the NT kernel, one can construct a framework that also solves the last problem, the event loop control-flow nightmare [15, 7, 5].

EXene [6] is a user interface toolkit for the X Windows windowing system [20], built on top of Concurrent ML, a concurrent language providing higher-order functions, static typing, lightweight threads and synchronous communication channels [18]. EXene uses a threading model from the grounds up, leading to a design both elegant and simplifying a lot of the difficulties typically encountered in user interface toolkits. It is important to note that eXene interacts directly via the X protocol, without relying on an underlying toolkit.

The goal of this paper is to isolate the difficulties in providing an eXene-style framework for programming Win32 applications. We focus specifically on the two following points: moving from an event loop model to a threaded model with channel-based communication, and decoupling the handling of system messages from the interaction with controls to help implement easily composable controls. The resulting system, although fairly conservative in its abstractions, is still much simpler to use than the raw Win32 API, and further supports the thesis that moving to a framework based on a high-level concurrent language leads to a simpler system with good modularity properties.

This paper is structured as follows: after reviewing the structure of Win32 programs, we describe Concurrent ML and give an outline of the framework, focusing on the important points of window management. We then describe how controls are handled, including predefined

controls and custom controls. We also describe how to handle ActiveX Controls within the same framework in as transparent a way as possible. We conclude with some discussion of related and future work.

We assume the reader has a passing knowledge of higher-order languages in general.

2 Win32 programming

We review in this section the fundamentals of Win32 programming, at the level of tutorial books such as [14]. The Win32 API is closely linked with a given program structure. A Win32 program has an entry point `WinMain`, whose role is to initialize the application by creating the various windows making up the interface. The program then goes in a loop, reading messages from its message queue and dispatching them to the appropriate window for processing.

Classes. Every window belongs to a class, which needs to be registered prior to being used. A class sets the default icons, cursors, background colors and menu for every window of that class. A class also contains a pointer to a window procedure, which is a function invoked every time a message is sent to the window. Since a window procedure is associated with a given class, this implies that every window of that class share the same window procedure.

Window procedures. A window procedure is called whenever a message is sent to an application, either by another window or by the system. Messages are sent when a window is created, moved, resized or destroyed, when the mouse moves over the client area of the window, when the mouse buttons are clicked, when a key is pressed and the window has focus, when the window needs repainting, when a timer expires, and so on. Windows provides default handling for all of these messages, but an application will want to deal with some of them to provide its functionality.

Child windows. To simplify the creation of user interfaces, it is possible to use child windows to subdivide the area of a window into more manageable components. Each child window has its own window procedure, thereby encapsulating the behavior of the child window and allowing a certain amount of abstraction. The child window can decide to only send a few digested messages back to its parent window, which can deal with them more easily than otherwise possible. Typical example of child windows include *controls*, such as buttons, scrollbars and edit controls.

That's all there is to Win32 programming, really. Everything else is concerned with the processing of messages and their arguments, to do things such as drawing in a window (by processing the `WM_PAINT` message), handling mouse movement (by processing the `WM_MOUSEMOVE` message), handling keyboard input (by processing the `WM_KEYDOWN` message). Handling controls such as pushbuttons and edit controls is also done through messages. A pushbutton, for example, notifies its parent window of an interesting event (e.g. it has been clicked) by sending a `WM_COMMAND` message to the parent, with a code identifying the control and the notification code as arguments.

3 Concurrent ML

The language we use to express our concurrent framework is Concurrent ML (CML) [18], a concurrent extension of the mostly-functional language Standard ML (SML) [11]. SML provides among other things higher-order functions, static typing, algebraic datatypes and polymorphic types. CML is provided as a library-style extension to SML, with the following (simplified) signature:

```
structure CML : sig
  type 'a chan
  type 'a event
  type thread_id
  val channel : unit -> 'a chan
  val spawn : (unit -> unit) -> thread_id
  val sendEvt : 'a chan * 'a -> 'a event
  val recvEvt : 'a chan -> 'a event
  val wrap : 'a event * ('a -> 'b) -> 'b event
  val choose : 'a event list -> 'a event
  val sync : 'a event -> 'a
end
```

CML is based on the notion of a *thread* which is concurrently executing thread of control. A function `spawn` is used to create a thread that evaluates a given function. Communication between threads is done via *channels*. Communication is synchronous: a send blocks until a receive reads the value on the channel, and vice versa. To help design abstract communication protocols, a first-class notion of *event* is introduced. An event decouples the communication capability of an operation from its actual execution (synchronization). Sending a value over a channel is actually a two-step process: we first create an event that says that the communication operation to be done will send a value over the channel, and

when we synchronize on that event, the value is sent over the channel. Synchronization blocks until the communication is performed. Basic event constructors include `sendEvt` and `recvEvt` for sending and receiving a value over a channel. Synchronization is performed by the `sync` operation.

Decoupling definition from synchronization allows for the building of combinators to describe more refined communication mechanisms. For example, given an event, the `wrap` operation wraps a function around that event creating a new event that behaves as follows: when you synchronize on the event, the original event is synchronized on, and the result of the synchronization is fed to the function which is evaluated. Given a set of events, you can also create a new event that is a non-deterministic choice over all those events with the `choose` operation: when you synchronize on the event, one of the original events is non-deterministically chosen and synchronized on. Finally, note that channels and events are *polymorphic* over the carried type (represented by the type variable `'a`): a channel of type `int chan` carries values of type `int`, and so on.

Concurrent ML is currently distributed with the Standard ML of New Jersey compiler [1].

4 The basic framework

We outline in this section our framework for programming Win32 user interfaces using the concurrency model provided by CML. The framework is built on top of a direct binding of the Win32 API in SML. An overview of the binding as well as examples of use are given in [16]. The binding was derived from an IDL description of the API using a tool for compiling IDL descriptions to SML code interfacing the described API [17]. This turned out surprisingly well[1] and allowed us to use code found in tutorial material such as [14].

For our framework, we build on top of the raw Win32 API some layers of abstractions that simplify and abstract away from many low-level details. We still remain very much in the spirit of Win32 however, in the sense that most functions are simply lifted from the underlying API. Abstractions are mainly concerned with replacing the event loop by an independent thread and allowing a more compositional treatment of controls.

The framework aims at supporting more or less the functionality described in the first volume of [10], along with various simplifying assumptions. This paper further simplifies matters for the purpose of presentation, and in order not to overwhelm the reader with superfluous and confusing details. Note that we only give the *signature* of the modules in this paper, that is the type of the operations and values provided by the various modules. Implementation details are not discussed.

Figure 1 presents the `Run` module, which is the main entry point of the framework. The main function of a program in our framework is a function of type

```
instance -> 'a
```

taking as argument the instance handle of the program and returning some type (exactly which type is returned is unimportant). This function will be in charge of creating the various windows of the application, and calling the message loop of the main window. The function `doit` of the `Run` module invokes the main function, supplying the application instance handle.

Various modules are provided that simply encapsulate some aspect of the API, lifting the functions without trying to generalize or abstract away some of the functionality. For example, Figure 2 and 3 present modules that deal respectively with icons and menus. Other modules such as `Cursor`, `Bitmap`, `Rect`, `Pen`, `DC` encapsulate different aspects of the API. All of these are fairly straightforward, and aside from their sheer number, their implementation does not offer any difficulties. It is definitely the case that future work should aim at finding new abstractions to reduce either the size or the complexity of this part of the framework.

The most important module from our point of view is the one that focuses on *window management*. Window management describes anything that relates to the manipulation of windows, including their creation, deletion, movement, as well as the management of the classes. As we saw, every window belongs to a class, that assigns a default icon, cursor and colors for every window of that class. Moreover, in raw Win32, the class also provides a window procedure to process messages to the window. The window procedure is shared amongst all windows of the class. It is not clear why this design was chosen. Informal explanations are given that this helps guarantee that every window of a given class can behave the same way. But since the window procedure upon reception

[1] There were some interesting issues raised in providing the Win32 API bindings — both in terms of support of the Win32 API in a strongly-typed setting, and in terms of the mapping from IDL to SML — that may or may not be described in a future article.

```
structure Run : sig
  type instance
  val doit : (instance -> 'a) -> 'a
end
```

Figure 1: The Run module

```
structure Icon : sig
  type icon
  val application : icon
  val hand : icon
  val question : icon
  val exclamation : icon
  val asterisk : icon
  val load : Run.instance * string -> icon
  val draw : DC.hdc * int * int * icon -> unit
end
```

Figure 2: The Icon module

```
structure Menu : sig
  type menu
  val load : Run.instance * string -> menu
  val get : Window.window -> menu
  val create : unit -> menu
  val createPopup : unit -> menu
  val appendItem : menu * int * string -> unit
  val appendPopup : menu * menu * string -> unit
  val destroy : menu -> unit
end
```

Figure 3: The Menu module

```
structure Window : sig
  type class
  type window
  datatype class_style = CS_HREDRAW
                       | CS_VREDRAW
                       | ...
  datatype window_style = WS_OVERLAPPEDWINDOW
                        | ...
  datatype show_style = SW_NORMAL
                      | ...
  val class : string * Run.instance * Cursor.cursor * Icon.icon * Brush.brush * class_style list -> class
  val unregister : class -> unit
  val create : class * string * window_style list * window option * int option * int option * int option *
               int option * Menu.menu option * Run.instance * (window * Msg.msg chan -> unit) -> window
  val createChild : class * string * window_style list * window * int option * int option *
                    int option * int * int * Run.instance * (window * Msg.msg chan -> unit) -> window
  val show : window * show_style -> unit
  val update : window -> unit
  val setForeground : window -> unit
  val move : window * int * int -> unit
  val getClientRect : window -> Rect.rect
  val destroy : window -> unit
  val send : window * Msg.msg -> unit
  val quit : int -> unit
  val msg_loop : window -> int
  val default : window * Msg.msg -> unit
end
```

Figure 4: The Window module

of a message also receives the handle of the window to which the message is addressed, it is very easy to write a window procedure that handles messages differently depending on the recipient of the message.

In our framework, we would like to have a thread replacing the window procedure, and actual messages over channels instead of the Win32 messages passed to window procedures. In order to keep messages lightweight, we would like to drop the requirement of passing the handle of the target window when a message is sent. Indeed, our function to send a message should extract the communication channel from the window type, and send the message to that channel, implicitly determining which window the message is sent to. To help this setup, we will have a thread assigned on a *per window* basis. Of course, one can still support shared processing amongst all windows of a given class by delegating every messages to a centralized thread that communicates with every window of a class.

Figure 4 presents an excerpt of the `Window` module containing the interesting parts of the code. Types are defined for classes, windows, and various style parameters for both classes and windows. A function `class` creates a class given the appropriate parameters, and automatically registers it. The functions `create` and `createChild` are used to create windows, given the class, title, optional owner window, position and size (a value of `NONE` for these forces the use of a default, equivalent to a `CW_USEDEFAULT` in raw Win32), optional menu, instance handle and a function to process messages. This last function is spawned automatically on its own thread and is passed the window being created and a channel to communicate with the window. A child window is similar, but instead of a menu it takes an integer that should uniquely identify the child window and that will be use to communicate with the parent window. Functions are then provided to show, move and destroy the window. A function `msg_loop` is used to initiate the message loop of a window[2]. A function `send` is used to send a message to a window. The function `quit` simply posts the `WM_QUIT` message in the message queue of the application, a requirement for exiting a message loop.

As an example, consider the following main function for an application that bounces a logo around a window. This example is taken from chapter 7 of [14], and is given in its entirety in Appendix A. It is as simple an initialization function as can be: only one class, a window created with mostly default values, and a simple message loop.

[2]This assumes that windows in the framework use standard message loops, a simplifying assumption.

```
fun winmain (instance) = let
  val c = Window.class
              ("BouncingSMLN", instance,
               Cursor.arrow, Icon.application,
               Brush.white,
               [Window.CS_HREDRAW,
                Window.CS_VREDRAW])
  val w = Window.create
              (c, "Bouncing SML/NJ",
               [Window.WS_OVERLAPPEDWINDOW],
               NONE, NONE, NONE, NONE, NONE,
               NONE, instance, bounce)
  val v = Window.msg_loop (w)
in
  Window.unregister (c);
  v
end
```

The module `Msg`, outlined in Figure5, defines the various messages that can be sent to windows by the system and by other windows through the `Window.send` function. There is a datatype constructor per message, and message parameters are automatically unfolded for easy retrieval and building. The function given to `Window.create` will be spawned and passed the newly created window and a newly created channel on which the thread will receive its messages. At this point, Win32 rules for processing messages apply: every message not processed by the application must be passed to default processing, which means invoking `Window.default` with the message as argument, and so on. Often, the thread will simply read from the input channel and process the messages, but it can also listen concurrently for events coming from other parts of the application or from controls.

Finally, although we will not discuss them here, we mention that most errors in Win32 functions get mapped to SML exceptions.

5 Controls

The first step in the creation of our concurrent framework for Win32 involved lifting window procedures into actual threads with which one can communicate using CML-style message-passing. We now turn to the second important aspect of our framework: compositional controls.

A control is "... a child window an application uses in conjunction with another window to carry out simple input and output (I/O) tasks." [10]. In reality, controls can achieve any level of complexity chiefly through composition: putting a bunch of controls together forms a bigger control with potentially a higher-level semantics. It is possible in raw Win32 to compose controls, but the amount of plumbing one has to write is mind-numbing.

Our aim is to make creating new controls by combining existing ones easy, while staying within the philosophy of Win32.

A requirement for this to work is that there be no difference between a predefined control (such as a pushbutton or an edit control) and a composed control. We also would like the communication to and from the control to be independent of the window procedure of the parent window. The basic idea is that a control will have a notification channel on which it communicates internal changes and interesting events. Communication to the control is achieved by invoking appropriate functions acting on the control.

5.1 Predefined controls

Many controls are predefined in Win32. These include various kind of buttons (push, check, radio), editing controls, list and combo boxes, scrollbars, and static controls. Providing them in our framework is fundamentally a matter of presenting them the right way to the user. For example, Figure 6 and 7 give the modules implementing respectively pushbuttons and edit controls. Note the similar format of the modules: both define a type for the control, a datatype defining the various notification messages that the control can report, a CML event that a thread can synchronize on to get the notification, functions to communicate with the control, a function to create the control, and a function to access the control as a normal window, allowing one to apply functions from the `Window` module.

The problem with such an interface is that it completely contradicts the default interface for controls implemented in raw Win32. A predefined control sends notifications directly to its parent window by sending a `WM_COMMAND` message to the window procedure, with its control ID as an argument and the notification as the other. What we want is to intercept that message and redirect it onto a CML channel.

One way to achieve this is for the system to to transparently create a child window around the control, which will be the parent of the control, in charge of capturing the `WM_COMMAND` messages and sending them onto a CML channel assigned when the control is created. All very straightforward, but some work is involved in making sure that all the messages sent to the control are communicated to the transparent child window. For example, applying `Window.move` to the control should move the control but also move the transparent child window, and similarly for resizes and most other window operations.

5.2 Custom controls

Custom controls are controls defined by the programmer. To create a new control, a programmer must determine the appearance of the control and its interaction with its subcontrols, if any, and its parent. The simplest example of a custom control is a layout control, which is in charge of maintaining the layout of its subcontrols according to some constraint criterion. Other more involved controls can include dozens of subcontrols interacting in a complex way. Dialog boxes can also be seen as a type of complex control.

By uniformity, we would like custom controls to respect the informal specifications given in the previous subsection. Technically, a custom control is a child window, created via the `Window.createChild` function. The thread associated with the window, in charge of handling messages to the window, defines the appearance of the control by handling the `WM_PAINT` message, and so on. Communication with subcontrols is achieved by listening for the notification events from the subcontrols, concurrently with handling messages for the window. Similarly, a channel for reporting notification events for the custom control needs to be allocated.

For example, a new control that encapsulates two pushbuttons might have a single notification message defined as:

```
datatype notify_msg = CLICKED of int
```

which simply reports which button has been clicked, and a controlling thread processing messages to the window that also listens to notification events from the two subcontrols and sends the appropriate notification when clicks occur (assuming a notification channel `notifyCh`, and pushbuttons `b1` and `b2`):

```
...
sync (choose ([wrap (recvEvt (ch), handle_message),
               wrap (PushButton.notifyEvt (b1),
                     fn (PushButton.BN_CLICKED) =>
                          send (notifyCh,CLICKED 1)
                      | _ => ()),
               wrap (PushButton.notifyEvt (b2),
                     fn (PushButton.BN_CLICKED) =>
                          send (notifyCh,CLICKED 2)
                      | _ => ())]
...
```

```
structure Msg : sig
  datatype msg = WM_SIZE of int * int
               | WM_PAINT of Rect.rect
               | WM_DESTROY
               | WM_TIMER of int
               | ...
end
```

Figure 5: The `Msg` module

```
structure PushButton : sig
  type push_button
  datatype notify_msg = BN_CLICKED
                      | BN_DOUBLECLICKED
  val notifyEvt : push_button -> notify_msg event
  val create : string * int * int * int * int * Run.instance -> push_button
  val windowOf : push_button -> Window.window
end
```

Figure 6: The `PushButton` module

```
structure Edit : sig
  type edit
  datatype notify_msg = EN_CHANGE
                      | EN_ERRSPACE
                      | EN_HSCROLL
                      | EN_KILLFOCUS
                      | EN_MAXTEXT
                      | EN_SETFOCUS
                      | EN_UPDATE
                      | EN_VSCROLL
  val notifyEvt : edit -> notify_msg event
  val getSel : edit -> (int * int)
  val setSel : edit * int * int -> unit
  val replaceSel : edit * string -> unit
  val canUndo : edit -> bool
  val emptyUndoBuffer : edit -> unit
  val undo : edit -> unit
  val create : string * int * int * int * int * Run.instance -> edit
  val windowOf : edit -> Window.window
end
```

Figure 7: The `Edit` module

Decoupling the logic of the communication with the subcontrols from the handling of system messages to the control greatly helps modularizing the code. Indeed, given a custom control, we could easily reuse the communication logic for some other control having the same "behavior", but maybe a wildly different appearance [9].

5.3 ActiveX Controls

No discussion of controls would be up-to-date without mentioning *ActiveX Controls* [3]. The ActiveX Controls technology goes back to *Visual Basic Extensions* (VBX), a mechanism for writing control components for use in the Visual Basic environment. These were generalized to *OLE Controls* for use in a general COM-based environment [19]. The main problem with OLE Controls is that they required the programmer to implement a large number of interfaces that had to be present for the control to be usable. This did not mix well with the lightweight requirement for downloadable controls over a network, and so ActiveX Controls were introduced, fundamentally OLE Controls with looser requirements.

ActiveX Controls are simply COM objects[3], and the support for ActiveX Controls in any framework is based on the corresponding support for COM objects. An application that can use ActiveX Controls is called a *control container*. The functionality of an ActiveX Control is divided into four parts (from [3]):

- providing a user interface;
- allowing the container to invoke the control's methods;
- sending events to the container;
- learning about properties of the container's environment and allowing the control's properties to be examined and modified.

As we discussed in [16], calling the methods of a COM object from SML is fairly easy. It is harder to make the framework into a control container, because that implies presenting the whole framework as a COM object with the appropriate interfaces that ActiveX Controls can access to communicate events. This is not impossible, but most implementations of SML do not allow this to be done easily. Given a suitable implementation of such a capability, it is not hard to see how ActiveX Controls fit in the above framework. Current work on the SML/NJ runtime system is in part aimed at solving this particular problem.

[3] They must also support self-registration.

6 Related work

The idea that concurrency helps in programming user interfaces is not new. Building on the original work of Squint [15] and Montage [7], eXene [6] exemplifies the consistent use of concurrency as a foundation for user interface construction [5]. More recently, Haggis [4], a functional framework built on top of a concurrent extension to Haskell, also demonstrated the usefulness of concurrency in such a context. However, as opposed to eXene and our approach, the model presented to the user is strictly sequential — concurrency is only used internally.

Compositionality of user interface elements is a requirement for a programmer-friendly toolkit. Systems such as Tk [13] are mostly based on the notion that a user interface is a widget (in our terminology, a control) composed of subwidgets. Building a user interface is a matter of composing the controls together in a hierarchical fashion. Tk however uses Tcl as its underlying language, and because of its lack of large-scale programming structures, it is not well suited to building large systems (although some large systems have indeed been built using Tcl/Tk). The basic ideas underlying compositionality are best presented from the point of view of the so-called Model-View-Controller approach, and we refer the reader to articles such as [9] for a deeper coverage of the issues.

Of course, another closely related system is the Microsoft Foundation Classes framework, which provide C++ classes structuring most of the Win32 API. MFC also allows the definition of methods to handle messages directly, removing the need to explicitly code up the window procedure. However, the model is still based on an event loop, and it is still hard to program computation-intensive applications that remain interactive. Kernel threads must be used to help manage the complexity. More experience with our system is required before further comparison can be made, especially with respect to the efficiency, maintainability and reuse possibilities of the code.

7 Conclusion

We have described in this paper the design of a simple concurrent framework for Win32 programming, based on a high-level concurrent language with lightweight threads. The description we have given is very much an outline, and indeed even our implementation is incomplete. We have not talked about color, dialog boxes, keyboard and mouse handling, multiple-document interfaces, floating menus, common dialogs, to name a few.

The important points about our framework are the move from an event loop foundation to a threaded model, and a decoupling of the processing of system messages from the notification messages from controls. This gives us a chance to derive easily composable controls. It also gives us a natural way to incorporate ActiveX Controls transparently into the framework.

Although the framework does not introduce a great many abstractions over the underlying Win32 API, the framework is still much easier to use than a raw Win32 system, and the resulting code more modular, thereby increasing reusability.

Future work. As we mentioned, the framework is quite simplistic, and does not go as far as it could go to abstract away from the underlying system. This was an experiment to try to impose a concurrent communication mechanism onto Win32 that supports an abstract view of controls decoupled from the window procedure, and nothing else. We tried to stay as close as possible to the raw Win32 programming style. Future work is planned in two directions. First, this project is but a first step in implementing a Win32 interface to Standard ML of New Jersey. The next step is the design of a real toolkit that can manage both X-windows and Windows (and eventually others), with an even more abstract notion of controls. An investigation into the use of reactive sublanguages [8] to express the logic behind the controls interactions in such a toolkit is also in the works. Second, we plan to investigate the feasibility of transferring some of this work to a C/C++ framework, perhaps at the cost of introducing a custom version of lightweight threads.

Acknowledgments. Thanks to John Reppy for many discussions relating to the subject of concurrency in user interfaces that led to the experiment described in this paper.

Availability. The Standard ML of New Jersey distribution is available from `http://cm.bell-labs.com/cm/cs/what/smlnj`, and information on the framework presented here can be found on the author's web page at `http://cm.bell-labs.com/cm/cs/who/riccardo`.

References

[1] A. W. Appel and D. B. MacQueen. Standard ML of New Jersey. In *Third International Symposium on Programming Languages Implementation and Logic Programming*, volume 528 of *Lecture Notes in Computer Science*, pages 1–13. Springer-Verlag, August 1991.

[2] J. Beveridge and R. Wiener. *Multithreading Applications in Win32*. Addison Wesley Developers Press, 1996.

[3] D. Chappell. *Understanding ActiveX and OLE*. Microsoft Press, 1996.

[4] S. Finne and S. Peyton Jones. Composing Haggis. In *Proceedings of the Fifth Eurographics Workshop on Programming Paradigms for Computer Graphics*. Springer-Verlag, 1995.

[5] E. R. Gansner and J. H. Reppy. A foundation for user interface construction. In B. A. Myers, editor, *Languages for Developing User Interfaces*, chapter 14, pages 239–260. Jones and Bartlett Publishers, 1992.

[6] E. R. Gansner and J. H. Reppy. A multi-threaded higher-order user interface toolkit. In Bass and Dewan, editors, *User Interface Software*, volume 1 of *Software Trends*, pages 61–80. John Wiley & Sons, 1993.

[7] D. Haahr. Montage: Breaking windows into small pieces. In *Proceedings of the USENIX summer conference*, pages 289–297. USENIX, June 1990.

[8] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1993.

[9] G. E. Krasner and S. T. Pope. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, August/September 1988.

[10] Microsoft Corporation. *Win32 Programmer's Reference*. Microsoft Press, 1993.

[11] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, Mass., 1997.

[12] B. A. Myers. Separating application code from toolkits: Eliminating the spaghetti of call-backs. In *ACM SIGGRAPH Symposium on User Interface Software and Technology*, 1991.

[13] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison Wesley, 1994.

[14] C. Petzold. *Programming Windows 95*. Microsoft Press, 1996.

[15] R. Pike. A concurrent window system. *Computing Systems*, 2(2):133–153, 1989.

[16] R. Pucella, E. Meijer, and D. Oliva. Aspects de la programmation d'applications Win32 avec un langage fonctionnel. In *Actes des Journées Francophones des Languages Applicatifs*, pages 267–291. INRIA, 1999.

[17] R. Pucella and J. H. Reppy. An abstract IDL mapping for Standard ML, 1999. In preparation.

[18] J. H. Reppy. CML: A higher-order concurrent language. In *Proceedings of the 1991 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 293–305. ACM Press, 1991.

[19] D. Rogerson. *Inside COM*. Microsoft Press, 1997.

[20] R. W. Scheifler and J. Gettys. The X window system. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.

A Bounce example

```
fun bounce (window,ch) = let
  val timerID = 1
  val rate = 20
  val moveR = 10
  val xTotal = 158
  val yTotal = 131
  val xRadius = 59
  val yRadius = 45
  fun onTimer (xS,yS,xC,yC,xM,yM,b) = let
    val hdc = DC.get (window)
    val hdcMem = DC.createCompatible (hdc)
    val _ = (Bitmap.select (hdcMem,b);
             DC.bitBlt (hdc,xC - (xTotal div 2), yC - (yTotal div 2),
                        xTotal, yTotal, hdcMem,0 , 0, DC.SRCCOPY);
             DC.release (window,hdc);
             DC.delete (hdcMem))
    val xC' = xC + xM
    val yC' = yC + yM
    val xM' = if (xC' + xRadius >= xS) orelse (xC' - xRadius <= 0) then ~xM else xM
    val yM' = if (yC' + yRadius >= yS) orelse (yC' - yRadius <= 0) then ~yM else yM
  in
    (xS,yS,xC',yC',xM',yM',b)
  end
  fun computeArgs (x,y,b) = (x,y,x div 2, y div 2, moveR, moveR, b)
  fun loop (args as (xS,yS,xC,yC,xM,yM,b)) =
    case (recv (ch))
      of Msg.WM_SIZE (x,y) => loop (computeArgs (x,y,b))
       | Msg.WM_DESTROY => (Timer.kill (window,timerID);
                            Bitmap.delete (b);
                            Window.quit (window,0))
       | Msg.WM_TIMER (t) => let
            val args' = if (t=timerID) then onTimer (args) else args
          in loop (args') end
       | m => (Window.default (window,m); loop (args))
  fun init () =
    case (recv (ch))
      of Msg.WM_CREATE => (Timer.set (window,timerIR,rate,NONE);
                           loop (0,0,0,0,0,0,
                                 Bitmap.load ("smlnj.bmp")))
       | m => init ()
in
  init ()
end
fun winmain (instance) = let
  val c = Window.class ("BouncingSMLN", instance,
                        Cursor.arrow, Icon.application,
                        Brush.white,
                        [Window.CS_HREDRAW, Window.CS_VREDRAW])
  val w = Window.create (c, "Bouncing SML/NJ",
                         [Window.WS_OVERLAPPEDWINDOW],
                         NONE, NONE, NONE, NONE, NONE,
                         NONE, instance, bounce)
  val v = Window.msg_loop (w)
in
  Window.unregister (c);
  v
end
```

FIFS: A Framework for Implementing User-Mode File Systems in Windows NT

Danilo Almeida
dalmeida@mit.edu

Laboratory for Computer Science
Massachusetts Institute of Technology

Abstract

This paper presents FIFS, a framework that facilitates file system research under Windows NT. FIFS addresses the high cost of file system development under Windows NT by providing a simple user-mode development environment. The environment is a Common Internet File System (CIFS) loopback server that seamlessly integrates with NT's Installable File System (IFS) architecture via the CIFS client included in the operating system. As such, it can provide full NT remote file system semantics. Initial performance measurements of the prototype FIFS implementation show FIFS capable of achieving good performance. Our prototype non-caching user-mode NFS implementation performs at about 70% the speed of a commercial non-caching kernel-mode NFS implementation.

1 Introduction

This paper presents FIFS, a user-mode framework for implementing file systems in Windows NT. For many years, file system research has been conducted on UNIX, which is popular in academic research environments. However, there has recently been increasing interest in Microsoft's Windows NT as a research operating system. Unfortunately, Windows NT does not a have a well-documented, inexpensive file system development environment. Its I/O architecture is radically different from the traditional UNIX architectures and is thus not as widely understood. FIFS is an effort to make it easy to write file systems for Windows NT.

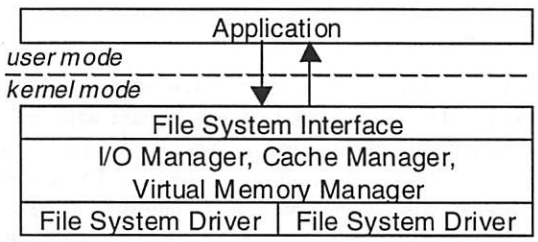


Figure 1-1: Overview of NT IFS Architecture

1.1 Background

The file systems shipped with Windows NT are implemented as kernel-mode file system drivers that plug into the NT's Installable File System (IFS) architecture. They interact with the NT I/O manager, cache manager, and virtual memory manager to satisfy file I/O requests as shown in Figure 1-1.

Unfortunately, writing file systems for Windows NT has been hard for a variety of reasons. As a kernel-mode device driver, a file system driver cannot access traditional operating system services available to user-mode programs. It must also comply with kernel paging restrictions. Due to NT's highly asynchronous I/O architecture, device drivers must not block and must be fully re-entrant. The driver is also hard to debug because it runs in a single address space with all other kernel-mode components and must be debugged with special kernel-mode debuggers rather than the generally more familiar user-mode tools. A file system driver is an especially difficult type of device driver to write. While traditional device drivers interact mostly with just the I/O manager, a file system driver must also engage in complex interactions with the cache manager and the virtual memory manager when satisfying file I/O requests.

While there is some support for writing general device drivers for NT, there is almost no support or documentation to aid in file system driver development. Microsoft has limited support for file system development via the NT Installable File System (IFS) Development Kit. At the beginning of 1997, Microsoft started selling the NT IFS Development Kit for US$1,000 [16]. The kit includes no documentation of the IFS architecture, but instead provides a single half-megabyte header file and source code for the FAT and CDFS[1] file systems. The sample code is fairly optimized and complex, and, without documentation, requires that developers reverse engineer the IFS architecture. In addition, there are no assurances that the file system-related portions of the driver environment will remain stable between releases of Windows NT [16, 17, 18]. In late 1997, the first book about NT file system development was published [17]. While the book provides a great deal of help in kernel-mode file system driver development, it does not make the development process easy or rapid. In fact, the author stresses that developing file systems under NT is a time-consuming process (more so than traditional operating systems, i.e., UNIX).

[1] CDFS provides ISO-9660 (CD-ROM file system) support in Windows NT.

1.2 Goals

Several important criteria affect file system development. Any file system development framework must address the following issues:

price - The framework should not impose a high additional cost barrier to file system development.

performance - File systems developed under the framework must perform at speeds close to traditional implementations.

portability - The framework and file systems should be easily portable across operating system revisions and perhaps even across different operating systems.

richness of semantics - File systems developed using the framework should support full operating system file system semantics. For example, if the operating system and file system both support byte range locking, a framework should allow users to take advantage of the byte range locking capabilities through the operating system's standard file system interfaces.

ease of programming - The file system development interface provided by the framework must be easy for programmers to use.

ease of use - A file system should be easy to use. Programs should not need to be recompiled or re-linked to take advantage of a new file system.

Our goal is to provide a file system development framework that addresses these issues in a way that makes it easy for developers to implement file systems. Thus, we prefer ease of programming rather than the absolute highest performance.

1.3 Solution

FIFS is a free experimental file system development framework that addresses all of these issues. It is the first file system development framework for Windows NT that is fully implemented in user-mode. By running in user-mode, FIFS addresses the issues of portability and ease of programming. Rather than forcing the developer to write to the more volatile kernel-mode programming environment, FIFS allows the developer to write a simple user-mode dynamic link library (DLL), using traditional operating system facilities, programming libraries, and development tools. As a user-mode driver, a FIFS driver is much easier to debug. It is thus very well suited for experimental file systems. A file system developed with FIFS is easy to use as it plugs transparently into the Windows NT namespace. It has the potential to perform all file system operations available to NT user-mode programs, including file locking, byte range locking, and directory notification. Our initial performance tests show that our user-mode NFS FIFS file system driver performs comparably to a commercial kernel-mode NFS implementation.

In addition to the file system development framework itself, FIFS includes the FSWIN32, FSMUNGE, and FSNFS user-mode file system drivers. FSWIN32 is a FIFS file system driver that makes Win32 file system API calls to provide access to whatever is accessible via the local machine. It is mainly intended as a testing and performance measurement tool. FSMUNGE is a pathname dissection filter driver that allows simpler file system driver implementations to plug directly into FIFS. FSNFS is a simple NFS file system driver that works with FSMUNGE.

1.4 Organization

In this paper, we examine related work, discuss the design of FIFS, describe some aspects of the prototype implementations of FIFS and the FSWIN32, FSMUNGE, and FSNFS file system drivers. We also investigate the performance of the initial FIFS implementation by comparing FSWIN32 to local file system access and comparing FSNFS to a commercial kernel-mode NFS implementation. Finally, we suggest future work that needs to be done with FIFS.

2 Related Work

The vnode interface originally developed by Sun is one of the most common mechanisms for adding file systems to UNIX [5, 9]. It is simple compared to the interfaces used by NT's IFS architecture and has enjoyed a fair amount of use in research file system implementations [4, 23]. Until recently, there has been no similarly simple interface to NT file system development. It is our hope that FIFS will help change this situation.

There are several commercial file system development kits for Windows NT. One such kit is Open Systems Resource's (OSR) File Systems Development Kit (FSDK), which is based on source code licensed from Microsoft and is considered to be an excellent NT kernel-mode file system development kit [18]. It provides wrappers that attempt to isolate the file system driver developer from the complexity of the NT kernel. Unfortunately, the kit is sold for US$95,000. According to OSR, the price of the FSDK reflects the current cost of NT file system driver development [20].

There are several kernel-mode file system drivers available to users in addition to those that are shipped with NT. Most are commercial NFS clients. Some of these include Intergraph's DiskAccess, Xlink's Omni-NFS Enterprise, FTP Software's InterDrive, and Hummingbird's NFS Maestro. Since FIFS was conceived, there has been one free kernel-mode implementation of ext2fs released.

User-mode file systems have also been implemented under Windows NT. In mid-1997, Galen Hunt developed a kernel-mode proxy driver at Microsoft Research. This proxy forwards I/O requests from the kernel into user-mode and allows for the development of user-mode file systems

[7, 8]. Hunt provides a library that is specifically geared towards writing file systems with his proxy. Hunt successfully implemented HTTP and FTP file system drivers using his proxy. One novel feature of Hunt's proxy driver is that the user-mode drivers are developed as Component Object Model (COM) objects. FIFS uses a similar object-oriented approach. At the time FIFS was developed, Hunt's proxy driver was not available to the general public. Today, however, it can be downloaded from Microsoft Research. We have not had time to evaluate the framework since it was released to the public.

An alternative approach is to use an existing network file system driver to forward file system requests to a user-mode file server via a standard network connection. This server services the request and sends the response back to the network file system driver via the connection. When the user-mode file server runs on the same machine as the client network file system driver, the server is called a *loopback server*. This was the approach used in creating a portable UNIX SFS (secure file system) client implementation in [12]. In that case, an NFS loopback server was used because most UNIX kernels contain an NFS client. While Windows NT does not include an NFS client, it does include a Common Internet File System (CIFS) client, which uses the Server Message Block (SMB) protocol[2]. A CIFS loopback server is particularly suited to implement file systems on NT because some of the SMB calls are exactly the same as NT system calls. An example of a CIFS loopback server file system implementation is Transarc's commercial AFS client for Windows NT.

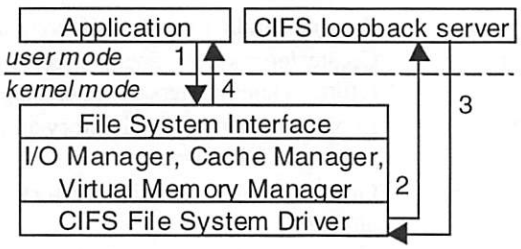


Figure 2-1: Operation of CIFS loopback server file system

Figure 2-1 shows how a CIFS loopback server file system implementation works. First, the application issues file system request to the kernel (1). Then, the kernel-mode CIFS client issues one or more corresponding SMB requests to the user mode CIFS loopback server (2). The user-mode CIFS loopback server fulfills the requests (possibly by going out over the network and/or reading the local disk) and returns results to the CIFS client as an SMB response (3). Finally, the application receives the results from the kernel.

A user-mode file system can also be implemented as a file system library. However, each program that wishes to make use of the new file system needs to be statically or dynamically linked to the library. Programs that use the file system interface exported directly by the kernel will simply be unable to use the new file system unless they are recompiled. No work on such a file system implementation has been done on NT. However, some work has been done in providing UNIX libraries on NT which provide users with UNIX-like semantics for file systems under NT [10, 25].

3 Design

We chose a user-mode CIFS loopback server[3] design for FIFS. This gives the framework full integration into the NT namespace as well as the potential for full NT remote file system semantics[4]. As a user-mode server using a standard protocol, the loopback server is portable across operating system revisions. To facilitate file system programming, the loopback server calls into a straightforward file system dispatch table.

3.1 Server

The loopback server is a user-mode process that listens on a NetBIOS name [14] and responds to CIFS requests from the local machine's CIFS client. (To prevent connection hijack attacks, requests are accepted only from the local client.) The requests can be divided into 2 categories: connection management and file system dispatch operations. Connection management requests consist of setting up the CIFS session to the local machine and performing user authentication. File system dispatch operations are the standard open, close, read, write, etc. operations.

When a user attempts to connect to a particular file system, the CIFS server can use pass-through authentication to the local machine. After establishing the identity of the user, the server passes the user's principal identifier to an underlying user-mode file system driver and receives a dispatch table for the user. The dispatch table is a set of file system functions (like read, write, etc.) with an associated user security context. Subsequent file system operations for that user are invoked through that dispatch table.

The functionality supported via the loopback server is only as rich as the functionality provided by the SMB protocol.

[2] The terms *CIFS* and *SMB* are used interchangeably throughout the rest of the paper. They refer to the same protocol. Servers (or clients) implementing the protocol are known as CIFS and SMB servers (or clients).

[3] For the remainder of this paper, the term *FIFS server*, *loopback server*, and *CIFS server* refer to the FIFS loopback server. Unless otherwise noted, other references to *server* also refer to the FIFS loopback server.

[4] The CIFS semantics are very similar to NT file system semantics. For details on their similarities, see [11] and [16].

Since SMB does not support such things as hard links, creating symbolic links, and setting standard UNIX ownership information and mode bits, the framework cannot support these features directly. However, this additional functionality can be provided over SMB via IOCTLs.

3.2 Namespace

A user names files on a FIFS file system via universal naming convention (UNC) names of the form `\\server\share\path`, where `server` is the NetBIOS name of the FIFS server and `share` can be anything. The NT CIFS client will direct requests to names of this form to the FIFS loopback server, passing the `share` identifier and `path` portion of the name to the server. A user can avoid having to specify a UNC pathname by mapping the `\\sever\share` portion of the pathname to a drive letter.

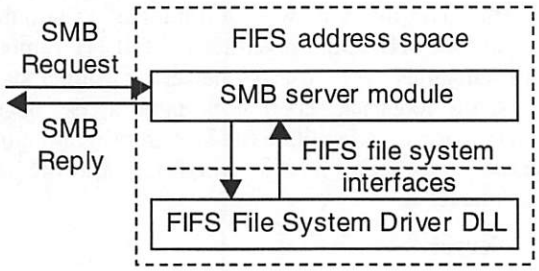


Figure 3-1 : FIFS Architecture

3.3 File System Drivers

The FIFS server uses user-mode FIFS file system drivers to satisfy SMB requests as shown in Figure 3-1. A file system driver is implemented in a Windows DLL. The DLL exports a single function, `FileSystemCreate()`, that allows the server to request an interface pointer to a `FileSystem` interface. `FileSystemCreate()` takes in a file system name, a configuration string, and an interface version. It returns a pointer to the desired version of the `FileSystem` interface for the corresponding file system, configured according to the configuration string. In pseudo-code, the function is defined as follows:

```
FileSystem = FileSystemCreate(fs_name,
        fs_config_path, version)
```

This design is based on the Component Object Model (COM) [3]. The `fs_name` argument allows a DLL to act as a driver for multiple file systems. The `fs_config_path` argument allows the file system to be dynamically configured by the server. If a new version of the file system interface is developed, a different `version` number can be used for the interface. Then, the server can try to use the latest version of the file system interface supported by the file system DLL.

3.4 File System Interfaces

The file system interfaces are thread-safe and provide COM-like reference counting. As thread-safe interfaces, they can be readily used in a multi-threaded loopback server. As in COM, reference counting is achieved via `AddRef()` and `Release()` methods in the interfaces. Programmers using these interfaces must therefore call `AddRef()` whenever they assign an additional reference to the interface and `Release()` whenever they release a reference to the interface object. This is so that the memory allocated for the interface can be automatically de-allocated by the interface object itself once its reference count is zero. Any function that allocates an interface object and returns an interface pointer (such as `FileSystemCreate()` above and `FileSystem::connect()` below) must ensure that the reference count for the interface is equal to one.

The initial version of the file system interfaces is `FS_VERSION_0`. The main interface is the `FileSystem` interface. Aside from reference counting, the only function that this interface provides is `connect()`, which, given a principal identifier for a user, returns a `FsDispatchTable` interface that is associated with the user's security context.

The `FsDispatchTable` interface is a simple, handle-based file system interface derived from the Win32 interface and the vnode interface. Aside from `AddRef()` and `Release()`, it contains the following functions:

Function	Description
`get_principal`	Returns principal associated with this dispatch table
`get_root`	Returns handle to root of file system.
`create`	Creates/opens files, opens directories and returns a handle, depending on flags.
`lookup`	Looks up a name in a directory and returns its attributes.
`set_attr`	Given a handle, sets file/directory attributes.
`get_attr`	Given a handle, returns file/directory attributes.
`close`	Closes a handle.
`write`	Writes data to a file handle at the specified offset.
`read`	Reads data from a file at the specified offset.
`read_dir`	Given a directory handle and cookie, returns directory entries.
`statfs`	Returns file system attributes, including volume name and size information.
`remove`	Removes file with given name from a directory.
`rename`	Renames a file.
`mkdir`	Creates a directory with the specified attributes.

`rmdir`	Removes a directory.
`readlink`	Given a symbolic link handle, returns path to which it points.
`symlink`	Creates a symbolic link.
`link`	Adds a hard link.
`ioctl`	Performs an IOCTL on a file handle.
`flush`	Returns after putting file on stable storage.

Table 3-1: Summary of FsDispatchTable interface

This interface allows all standard directory and file operations to be performed. However, it does lack locking and callback notification facilities. Section 6.3 discusses these missing features in more detail.

3.5 Layering: Filters and Adapters

The framework provided by FIFS is flexible. A file system driver can call into any other file system driver in the same way that the server can. Thus, the framework can achieve layering of file system drivers. Figure 3-2 illustrates this principle.

There are a wide variety of applications for file system driver layering. For example, a simple encrypting layer can be used as a filter to encrypt and decrypt all data written to and read from a given file system. A separate filter driver might audit all file accesses in a particular directory that contains confidential information.

A file system driver can also be used as an adapter from the FIFS server component to a simple file system implementation. For example, a simple FIFS server can be implemented such that it is not aware of symbolic links and never calls `readlink()` and `symlink()`. To make this server work with a file system driver that returns symbolic link attributes and does not automatically traverse symbolic links, an adapter layer can be written that transparently traverses symbolic links. Similarly, if a file system driver is case sensitive, but the names provided by the CIFS client are not, a file system adapter can mask the mismatch between the server and the client.

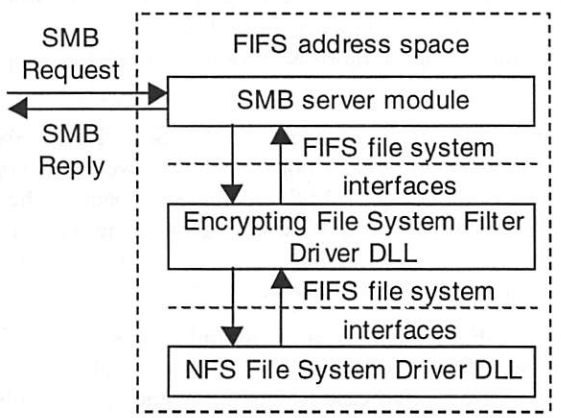


Figure 3-2 : Example of File System Layering in FIFS

This layered architecture allows developers to write simple file system drivers that can be matched to whatever server implementation is being used through any number of layered drivers. Thus, if desired, a filter or adapter can be simple and do a single task but several can be combined to perform complex data transformations and actions.

4 Implementation

The initial FIFS implementation consists of the CIFS loopback server, two file system drivers, and an adapter driver.

4.1 Server Implementation

The SMB protocol used in CIFS supports several different dialects [11]. The first step in implementing the FIFS loopback server is the selection of an SMB dialect. The newer dialects have more advanced functionality (e.g., file locking, better user authentication, etc.) but also support all requests in the older dialects. The CIFS specification suggests that new clients using a new protocol should not use older-style SMB messages so that, in the future, new SMB servers will not have to support the older messages. However, new SMB servers are currently supposed to support old-style messages in their dialect. This makes writing SMB servers more cumbersome than necessary.[5] In this implementation, the `LM1.2X002` dialect [11] is used. It provides the richest semantics without the more obscure NT-specific features of the more recent `NT LM 0.12` dialect [11]. Because all current SMB dialects include older dialects, an `LM1.2X002` implementation can be used as a stepping stone to an `NT LM 0.12` implementation.

A drawback of not using `NT LM 0.12` is that IOCTLs are not supported in previous SMB dialects (or, at least, are undocumented). Therefore, this FIFS implementation does not support IOCTLs.

For the initial FIFS implementation, we chose not to support SMB's opportunistic locking [11] so as to simplify the implementation. We also do not implement pass-through authentication. Sections 6.1 and 6.3 have more details on these topics.

The server is multi-threaded and maintains a minimal amount of global state that is protected from concurrent access. (This state mainly consists of `FsDispatchTable` pointers.) The access operations are fast and allow the server to be highly concurrent. The only time-consuming operations that a thread might do are calls into a dispatch table. In that case, the file system driver is responsible for

[5] The correct solution is to define a new dialect that only supports the newer-style SMB messages. Then, writing servers that speak the newest SMB dialect would be a less cumbersome task. Since the SMB loopback server needs to work with the current NT CIFS client, it cannot define a new SMB dialect and must instead use one of the supported dialects.

safely maintaining its internal state and achieving as much concurrency as desired.

The server does not directly call any of the symbolic or hard link functions in underlying drivers because SMB does not know about symbolic links. Instead, it relies on the file system driver to provide transparent access to symbolic links (directly or via a layered driver).

4.1.1 Configuration

Server configuration parameters are specified via a Windows NT registry pathname argument to the server. The configuration includes NetBIOS name (which by default consists of the local machine name and a few extra characters), the desired NetBIOS buffer size, the number of worker threads desired, the file system driver DLL to use, the file system name to ask for, and the file system configuration information string. The server currently runs as a regular process rather than as a Windows NT service.

4.1.2 Pathnames

An important feature of this server is that it passes pathnames to the underlying file system driver without interpreting them. Since such pathnames are often absolute pathnames, the underlying file system driver must be prepared to handle a backslash-delimited pathname. An advantage of this approach is that the FIFS server does not have to split up the name and traverse the pathname by calling `create()` multiple times. Rather, it can just pass the full name to the underlying file system driver, which can do whatever it wants to do.

One problem that we discovered while implementing the server is that the NT CIFS client sometimes passes uppercase pathnames to the loopback server. This is a problem if the underlying FIFS file system driver is case-sensitive. In some cases, the NT client requests all entries in a directory from the loopback server. However, there are cases where the NT client passes an uppercase string as a filter to an SMB directory enumeration request. Our loopback server optimizes lookups for such a filter by calling the `lookup()` function on the given name instead of calling `read_dir()` and filtering the results. One problem with this approach is that `lookup()` does not return a name. So, the server fills in the name information in the SMB directory enumeration reply with the uppercase string that it received from the CIFS client. This can be a problem if the client then uses the name to open a file on a case-sensitive file system. In order to circumvent this idiosyncrasy, some future work can be done either in the main file system driver itself or as a layered driver (see Section 6.6).

4.2 FSWIN32

FSWIN32 is the first file system driver implemented for FIFS. It allows the user to access a subtree of the local machine's namespace. It simply converts its arguments and calls directly into the Win32 API. The implementation uses coarse locking and thus exhibits little concurrency. Most of the file system functions in this driver lock the user's entire dispatch table object. The purpose of this file system driver was to do initial framework validation as well as to get an idea of the overhead of FIFS when accessing parts of the Win32 namespace. Its only interesting performance feature is that it pre-fetches and caches directory information.

4.3 FSMUNGE

FSMUNGE is a file system adapter. Its configuration information specifies the underlying file system driver to which it will serve as an adapter. Whenever FSMUNGE receives a request with a multi-part pathname, it parses the pathname and opens each directory component using the underlying file system driver. It then fulfills the request by calling the underlying file system driver with the resulting directory handle and final pathname component. This filter driver is fully asynchronous. It will block only if the underlying file system driver blocks.

The purpose of this file system adapter was to allow us to develop FSNFS without having to handle multi-part pathnames. FSMUNGE was easy to develop. In fact, it only took about an hour of development time, most of which was spent writing a pathname dissection class. The ease of development is a good indication of how easy it is to write FIFS drivers. FSMUNGE was later revised to make pathnames lower case so that the underlying FSNFS driver did not have to handle uppercase names sent by the NT CIFS client. (A side effect of this is that the FSNFS driver is unable to access files with names containing upper case letters. A fix to this problem is suggested in Section 6.6).

4.4 FSNFS

FSNFS is an NFS version 2 file system driver. Like FSMUNGE, this driver took relatively little time to develop. Its was implemented in a day by someone who had never implemented an NFS client or server and who had never written a significant piece of code using the ONC/Sun RPC library. Due to lack of time, we did not add any caching optimizations to this driver. So FSNFS must always make one or more NFS remote procedure calls to satisfy requests.

Because the freely available ONC/Sun RPC library implementation for NT is not thread-safe, we use a coarse locking discipline for FSNFS. One big drawback to the low FSNFS concurrency is that large read and write requests get broken down into 8KB chunks that are issued synchronously rather than asynchronously.

Though NFS supports symbolic links, this first FSNFS implementation does not. FSNFS does not make any provisions to handle case-insensitive names passed into it. The driver could support this functionality by reading the directory where the name lookup is taking place and doing

a case-insensitive match. Since FSNFS does no caching, such functionality can just as easily be implemented as an adapter driver. Section 6.6 discusses the issues that must be addressed to fix this problem.

5 Experiments and Results

We compared the performance of accessing the local NTFS file system directly versus access through FSWIN32 to obtain a rough measurement of the overhead of FIFS. We also compared FSNFS to NFS Maestro Solo, a commercial NFS client implemented as a kernel-mode file system driver created by Hummingbird Communications Ltd. We chose NFS Maestro because it has been rated as one of the best-performing NFS clients for NT. By default, Maestro uses the NT cache manager. However, it does have an option to disable caching. We believe that it may be possible to integrate FIFS with the NT cache manager (see Section 6.3). However, this initial implementation of FIFS does not include cache manager integration. To obtain a clearer picture of the FIFS overhead, we ran Maestro both with and without caching. We ran the LFS large and small file micro-benchmarks to help understand the performance characteristics of these systems. We then ran an application benchmark so as to understand how these file systems compare under a particular type of workload.

5.1 Setup

The FIFS loopback server and NFS server for these experiments were both 200MHz Pentium Pro machines with 256KB L2 cache, 64MB RAM, 2GB disks, and 10Mb/s SMC Ultra Ethernet cards connected via a 100Mb/s switch. The FIFS machine ran Windows NT Server 4.0 with Service Pack 3 while the NFS server ran OpenBSD 2.2. The tests were run 3 to 5 times to verify that they generated similar results. Neither machine was loaded. While no special care was taken to isolate the machines from broadcast traffic and other connections, the network was monitored to verify that the tests ran under similar conditions. The loopback server was configured to run with a 16KB buffer size. We configured NFS Maestro to use NFS version 2. In addition, as per its performance optimization configuration tool, NFS Maestro was configured to use a 4KB read size and an 8KB write size with no parallelism on reads and 8-way parallelism on writes.

In the results, we indicate the FSWIN32 driver running in a single FIFS server worker thread as FSWIN32-1. Similarly, the FSNFS driver running in a single worker thread is indicated by FSNFS-1. FSWIN32-4 and FSNFS-4 indicate four-threaded runs of the FIFS server with each of these drivers. The kernel-mode NFS Maestro results are listed as kNFS. kNFSnc indicates the results of running kNFS without caching enabled.

5.2 Large File Micro-Benchmark

The large file benchmark sequentially writes a large file, reads it sequentially, writes it randomly, reads it randomly, and then sequentially re-reads it. Data is flushed to disk after each write. For these tests, we used an 8MB file with I/O sizes of 256KB and 8KB. The results are shown in Figure 5-1 and Figure 5-2.

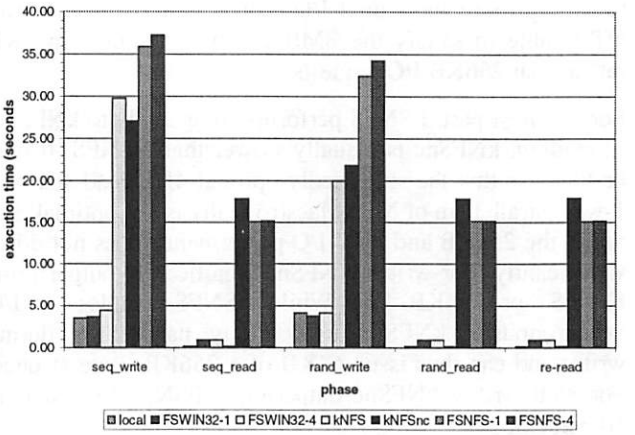


Figure 5-1 : Large File Micro-Benchmark Results (8MB file using 256KB I/O)

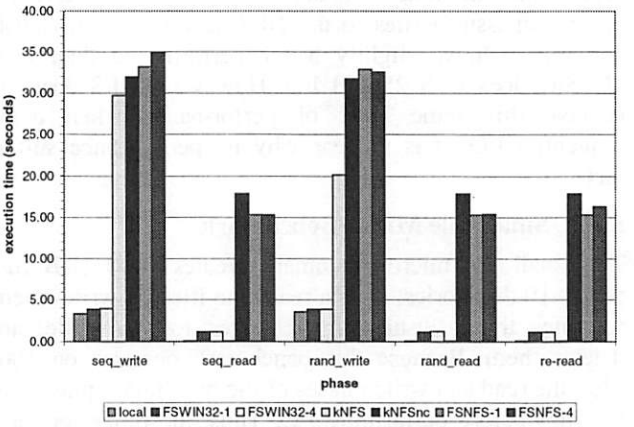


Figure 5-2 : Large File Micro-Benchmark Results (8MB file using 8KB I/O)

The benchmark shows that the FIFS file system driver implementations are slightly slower when running with multiple worker threads. This is expected as the current FSWIN32 and FSNFS implementations block the entire file system dispatch table on each call.

The large file write performance for FSWIN32 versus direct NTFS access shows that FIFS does not add much overhead to writes. We are uncertain as to why FSWIN32 exhibited slightly better performance in the random write phase of the benchmark. Varying the I/O size does not significantly affect the FSWIN32 results.

FSWIN32 read performance is poor compared to direct NTFS access. The difference is 0.87 seconds for FSWIN32 versus 0.15 seconds direct NTFS using 256KB I/O and 1.14 seconds versus 0.07 seconds for 8KB I/O. This is because the NT I/O subsystem cannot satisfy FSWIN32 read requests by looking directly at the NT cache. Instead, NT must use FIFS to satisfy the request. FSWIN32 takes more time for the 8KB I/O than the 256KB I/O because it needs to satisfy more individual I/O requests. It is unclear why NT is able to satisfy the 8MB I/O more quickly in 8KB rather than 256KB I/O requests.

For the most part, FSNFS performs comparably to kNFSnc. In reading, kNFSnc is actually slower than FSNFS. It may be the case that the supposedly optimal 4KB read size and 1-way parallelism of NFS Maestro really is not optimal. For reads, the 256KB and 8KB I/O performance does not differ significantly. For writes, kNFSnc significantly outperforms FSNFS for 256KB I/O. While FSNFS handles its I/O synchronously, kNFSnc uses 8-way parallelism during writes and can thus issue 64KB of a 256KB write at once. For 8KB writes, kNFSnc outperforms FSNFS by less than 10 percent.

With NT cache manager integration enabled, kNFS is significantly faster than FSNFS. In reads, kNFS is as fast as NTFS since the data is in the NT cache. Unlike kNFSnc, kNFS can issue writes to the NFS server asynchronously and thus achieve slightly better performance than even kNFSnc does with 256KB I/O. However, kNFS does not achieve this same level of performance when doing sequential I/O. It is unclear why its performance suffers there.

5.3 Small File Micro-Benchmark

The small file micro-benchmark creates 1000 1KB files across 10 directories. It then reads the files, re-writes them, re-writes them flushing the changes for each file, and deletes them. Because this benchmark operates on 1000 files, the read and write phases of the benchmark must open the files before performing I/O. Thus, the times for these benchmarks reflect the time to lookup each file. Figure 5-3 shows the benchmark results.

FSWIN32 performance for create, read, and both types of write takes an additional constant amount of time compared to direct NTFS access. This is because the SMB reply to the opening of a file includes some file attribute information. So, FIFS needs to get attributes for each of the 1000 files that are opened. For delete, however, there is no such overhead, so the execution times are nearly the same.

For the NFS clients, the small I/O size prevents I/O overlap while the lookups and deletes synchronize access to the NFS server. FSNFS performance is not as good as kNFS and kNFSnc performance because FSNFS does a lookup on every pathname component when a file is opened. kNFS and kNFSnc cache the directory lookups and thus save the lookup times. Even with some network tracing, we are unable to explain why the read difference is large while the write and delete differences are small. A more detailed study is needed.

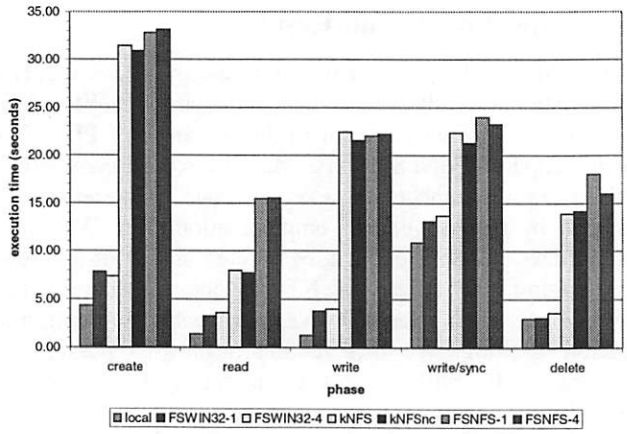


Figure 5-3 : Small File MicroBenchmark Results (1000 1KB files in 10 directories)

5.4 Application Benchmark

Our application benchmark represents a program build scenario. The source tree built in the benchmark is an older version of the FIFS source tree. It contains 209 files with an average file size of approximately 4.5KB.

The benchmark first copies a zip file containing the source tree. It then unzips the source tree and copies it into a new directory tree. It recursively checks the size of every file in the source tree using `du`. Next, it compares the two trees using a recursive `diff`. It then builds the source tree and recursively checks the size of the built source size using `du`. Next, it compares the built source tree with the original copy using a recursive `diff`. The build tree is then zipped into a new archive. It is also zipped into the original zip archive. Then, the trees and original archive are removed.

The benchmark results are divided into 6 categories: copy, unzip, attributes, compare, compile, zip, and remove. The copy, unzip, compare, compile, zip, and remove categories consist of the corresponding operations above. The attributes category consists of the `du` operations. The results are summarized in Figure 5-4 and Figure 5-5.

Looking at the total time shows that the FSWIN32 driver was overall not substantially slower than direct NTFS access. However, a big component of the test is the compilation phase, which has a high CPU utilization. The less CPU-intensive phases show FSWIN32's performance to be 2 to 4 times slower than direct access. The large file micro-benchmark suggests that these performance differences are due to reads rather than writes (see Section 5.2). The NT cache also caches directory information, so FSWIN32 suffers just as much when reading directory information as when reading data compared to direct NTFS

access. If it is possible to support the NT cache through support for CIFS opportunistic locks (see Section 6.3), FSWIN32 may become more on par with direct NTFS access.

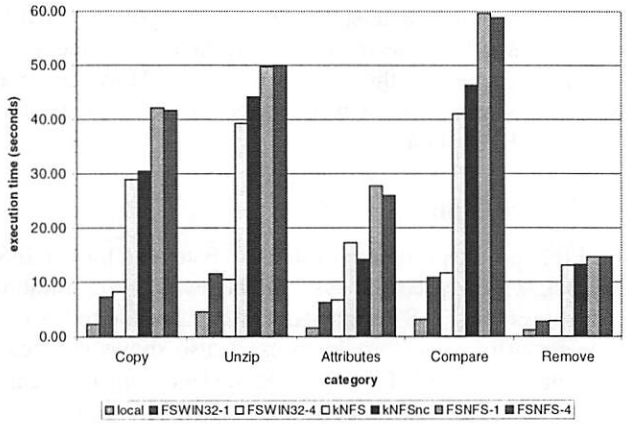


Figure 5-4: Application Benchmark Results – Part 1

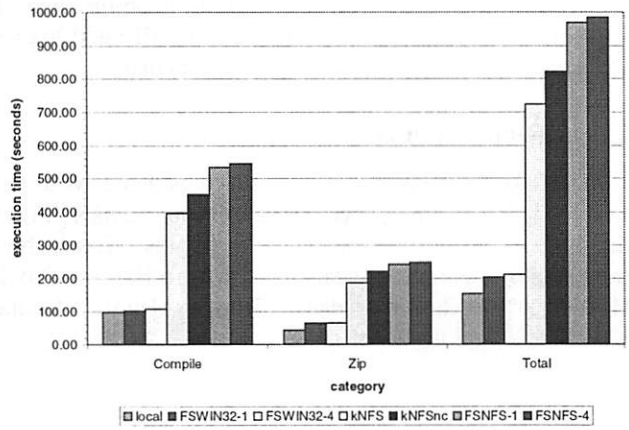


Figure 5-5: Application Benchmark Results – Part 2

The FSNFS driver performs reasonably well compared to NFS Maestro. The kernel-mode client performs at about 1.3 times the speed of FSNFS. The slowest application benchmark category for FSNFS is the attributes category. NFS Maestro is 1.6 times faster than FSNFS in that category. The problem is that FIFS and FSNFS interact poorly when reading directory entries. Section 6.4 contains a more detailed explanation of this problem and some possible solutions.

6 Future Work

While the current FIFS implementation does allow file systems to be implemented under NT, it is still missing some important functionality. The areas that need work are user authentication, IOCTL support, locking, `read_dir()` caching, symbolic link support, and achieving more concurrency in current FIFS file system drivers. We discuss each in turn.

6.1 Authentication

The server currently performs no real authentication and is thus unsuitable for multi-user use under NT. It should be straightforward to implement the pass-through authentication scheme described in Section 3.

6.2 IOCTLs

In order to support IOCTLs for non-CIFS file system semantics, the FIFS server needs to be updated to support the `NT LM 0.12` SMB dialect. This will require some additional work to implement the additional `NT LM 0.12` SMB messages. However, given the current `LM1.2X002` implementation, the change will be incremental in nature.

6.3 Locking, Change Notification, and Caching

FIFS currently has no support for locking and file and directory change notification. It currently lacks the necessary interfaces to support this functionality. In addition, the server currently lacks the support for SMB opportunistic locks that would be necessary to support such functionality. For locking and notification to be implemented, a callback needs to be passed into the file system so that the file system driver can notify the server of directory and file changes. This would require the addition of some extra lock status state to the server and some additional notification threads to the server and file system drivers. The callback mechanism would also support layering as filter drivers could store a higher level driver's callback and pass its own call back to the underlying file system driver.

If this functionality is added, the framework will be able to efficiently support caching of callback and lease-based network file systems like AFS and SFS. With opportunistic locks enabled, the CIFS client will be allowed to cache files directly and will not need to go the loopback server on cache hits.

One of the most important areas to investigate in this area is whether the NT CIFS client will take advantage of the server's opportunistic lock support and use the NT cache manager to cache data. If so, it may be possible to have a FIFS file system driver whose performance more closely matches kernel-mode file systems that use the NT cache manager.

6.4 Reading Directories

Some investigation of the network traffic indicates that the reading of directory entries could be made more efficient. Currently, a file system that does not cache directory entry information for an open directory (such as the current implementation of FSNFS) can suffer from unnecessary directory read overlap. The problem is that the size of each entry in the SMB directory enumeration response message

depends on the length of the returned file names. Thus, the loopback server needs to know how many entries to read from the underlying file system driver to fill the SMB response buffer. Currently, the server will make a guess and keep calling the underlying file system driver until it fills up the response buffer. So, the loopback server ends up ignoring extra directory entries from the file system and must re-read them when the CIFS client asks for more directory entries. FSWIN32 does not suffer from this behavior because it caches directory responses. However, FSNFS goes out to the network each time.

The only way to fix this is to cache extra directory entries that get returned. This cache can be added at either the file system driver level (like FSWIN32) or the server level. While a server level cache would benefit all file system drivers, it might be redundant if a file system does its own caching across different directory enumeration operations. Therefore, it may be worthwhile to write a simple filter driver that caches directory entries during directory enumeration. Then the filter driver can be used with file system driver implementations that do not do their own caching of directory enumeration information.

6.5 Symbolic Links

Symbolic links were not explored in any of the current FIFS file system drivers. To validate the framework's ability to easily deal with symbolic links, a simple symbolic link adapter driver should be developed to make symbolic links transparent to the CIFS loopback server. With this adapter, UNIX-style file system driver implementations such as FSNFS would simply have to return symbolic link attributes and implement `readlink()` and `symlink()` to provide transparent symbolic link support.

6.6 Case-Sensitive File Systems

The current FIFS framework does not transparently handle case-sensitive file systems (see Sections 4.1.2 and 4.3). This issue can be addressed via a filter driver that reads the directory where a case-insensitive name is being looked up and does a case-insensitive string compare to figure out the corresponding case-sensitive name. The filter can then pass the request through the underlying driver with the case-sensitive name.

Such an implementation is fairly naïve, however. It could suffer from poor performance. A high performance implementation could maintain a per-directory cache of directory entries for recently accessed directories. The cache could be kept up-to-date via the directory notification callback suggested in Section 6.3.

6.7 Achieving More Concurrency

The performance of initial file system driver implementations for FIFS could be improved through fine-grained locking. While the server framework can achieve high concurrency through the use of multiple threads, the current file system driver implementations cannot. It should be possible to retrofit FSWIN32 with finer locking without too much difficulty. This should allow FSWIN32 to have better large file performance for small I/O sizes. FSNFS would likely benefit a lot more from concurrency than FSWIN32 running against the local file system. FSNFS would be able to issue multiple I/O requests on the wire without waiting for the server to reply. However, for FSNFS to achieve this, it would need to use a thread-safe concurrent RPC library.

7 Conclusion

The FIFS prototype demonstrates the potential for a CIFS loopback server-based file system framework. While initial work shows that FIFS performance is comparable to kernel-mode performance in certain cases, it also shows that read performance in the FIFS prototype suffers from not being able to use the NT cache manager. The short time required to implement the FSMUNGE and FSNFS file system drivers shows that FIFS provides a framework where file systems can be easily developed. It is our hope that further work on FIFS and its file system drivers will yield higher performance and a more functional implementation.

Acknowledgements

I would like to thank everyone who helped me write the thesis that led to this paper, especially my advisor, Frans Kaashoek, and my colleagues and friends, Constantine Sapuntzakis, George Candea, Massimiliano Poletto, David Mazières, Derek Truesdale, Russel Hunt, Joanne Mikkelson, and Luis Sarmenta.

Bibliography

[1] Art Baker. *The Windows NT Device Driver Book: A Guide for Programmers*. Prentice-Hall, Upper Saddle River, New Jersey, December 1996.

[2] Richard Black. Report on the development, structure and performance of ATM device drivers for the personal computer operating systems Windows NT, and Linux. In *ATM Document Collection 4 (The Green Book)*. University of Cambridge, September 1995.

[3] Kraig Brockschmidt. *Inside OLE, 2nd. ed.* Microsoft Press, Redmond, 1995.

[4] David K. Gifford, Pierre Jouvelot, Mark Sheldon, and James O'Toole. Semantic file systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 16-25, ACM, 1991.

[5] Berny Goodheart and James Cox. *The Magic Garden Explained: The Internals of UNIX System V Release 4*. Prentice Hall, 1994.

[6] John S. Heidemann and Gerald J. Popek. File-system development with stackable layers. *ACM Transactions on Computer Systems*. 12(1):58-89, 1994.

[7] Galen Hunt, Creating user-mode device drivers with a proxy . In *Proceedings of the USENIX Windows NT Workshop.* USENIX, 1997.

[8] Galen Hunt, Proxy Driver Home Page, from http://www.research.microsoft.com/os/galenh/proxy/, September 1997.

[9] S. R. Kleiman. Vnodes: an architecture for multiple file system types in Sun UNIX. In *Proceedings of the USENIX 1986 Summer Conference.* USENIX 1986.

[10] David Korn. UWIN – UNIX for Windows. In *Proceedings of the USENIX Windows NT Workshop.* USENIX, 1997.

[11] Paul Leach, Dilip Naik. A Common Internet File System (CIFS/1.0) Protocol. Internet-Draft, IETF, March 1997.

[12] David Mazières. Security and decentralized control in the SFS global file system. Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science Master's Thesis, August 1997

[13] Microsoft Corporation. Microsoft Windows-based Terminal Server, from http://www.microsoft.com/ntserver/guide/hydra.asp, December 1997.

[14] Microsoft Corporation. Platform Software Development Kit. *Microsoft Developer Network Library*, October 1997.

[15] Microsoft Corporation. Windows NT 4.0 Device Driver Kit. *Microsoft Developer Network Library*, July 1997.

[16] Microsoft Corporation. Windows NT IFS Development Kit. from http://www.microsoft.com/hwdev/ntifskit/, June 1997.

[17] Rajeev Nagar. *Windows NT File System Internals.* O'Reilly & Associates, September 1997

[18] Open Systems Resources, Inc. Open Systems Resources File Systems Development Kit, from http://www.osr.com/develkit/fsdk.htm, July 1997.

[19] Matt Pietrek. A Programmer's Perspective on new system DLL features in Windows NT 5.0, Part I. *Microsoft Systems Journal*, November 1997. (also available at http://www.microsoft.com/msj/1197/nt5dll.htm)

[20] Daniel Root, Open Systems Resources, Inc., from e-mail exchange, July 1997.

[21] R. Srinivasan. RPC: Remote Procedure Call Protocol Specification Version 2. RFC 1831, Network Working Group, August 1995.

[22] R. Srinivasan. XDR: External Data Representation Standard. RFC 1832, Network Working Group, August 1995.

[23] David C. Steere, James J. Kistler and M. Satyanarayanan. Efficient User-Level File Cache Management on the Sun Vnode Interface. In *Proceedings of the USENIX 1990 Summer Conference.* USENIX 1990.

[24] Sun Microsystems, Inc. NFS: Network File System protocol Specification. RFC 1094, Network Working Group, March 1989.

[25] Stephen Walli. OpenNT: UNIX application portability to Windows NT via an alternative environment subsystem. In *Proceedings of the USENIX Windows NT Workshop.* USENIX, 1997.

Detours: Binary Interception of Win32 Functions

Galen Hunt and Doug Brubacher
Microsoft Research
One Microsoft Way
Redmond, WA 98052
detours@microsoft.com
http://research.microsoft.com/sn/detours

Abstract

Innovative systems research hinges on the ability to easily instrument and extend existing operating system and application functionality. With access to appropriate source code, it is often trivial to insert new instrumentation or extensions by rebuilding the OS or application. However, in today's world of commercial software, researchers seldom have access to all relevant source code.

We present Detours, a library for instrumenting arbitrary Win32 functions on x86 machines. Detours intercepts Win32 functions by re-writing target *function images. The Detours package also contains utilities to attach arbitrary DLLs and data segments (called* payloads*) to any Win32 binary.*

While prior researchers have used binary rewriting to insert debugging and profiling instrumentation, to our knowledge, Detours is the first package on any platform to logically preserve the un-instrumented target function (callable through a trampoline) as a subroutine for use by the instrumentation. Our unique trampoline design is crucial for extending existing binary software.

We describe our experiences using Detours to create an automatic distributed partitioning system, to instrument and analyze the DCOM protocol stack, and to create a thunking layer for a COM-based OS API. Micro-benchmarks demonstrate the efficiency of the Detours library.

1. Introduction

Innovative systems research hinges on the ability to easily instrument and extend existing operating system and application functionality whether in an application, a library, or the operating system DLLs. Typical reasons to intercept functions are to add functionality, modify returned results, or insert instrumentation for debugging or profiling. With access to appropriate source code, it is often trivial to insert new instrumentation or extensions by rebuilding the OS or application. However, in today's world of commercial development and binary-only releases, researchers seldom have access to all relevant source code.

Detours is a library for intercepting arbitrary Win32 binary functions on *x86* machines. Interception code is applied dynamically at runtime. Detours replaces the first few instructions of the *target function* with an unconditional jump to the user-provided *detour function*. Instructions from the target function are preserved in a *trampoline function*. The trampoline function consists of the instructions removed from the target function and an unconditional branch to the remainder of the target function. The detour function can either replace the target function or extend its semantics by invoking the target function as a subroutine through the trampoline.

Detours are inserted at execution time. The code of the target function is modified in memory, not on disk, thus facilitating interception of binary functions at a very fine granularity. For example, the procedures in a DLL can be detoured in one execution of an application, while the original procedures are not detoured in another execution running at the same time. Unlike DLL re-linking or static redirection, the interception techniques used in the Detours library are guaranteed to work regardless of the method used by application or system code to locate the target function.

While others have used binary rewriting for debugging and to inline instrumentation, Detours

is a general-purpose package. To our knowledge, Detours is the first package on any platform to logically preserve the un-instrumented target function as a subroutine callable through the trampoline. Prior systems logically prepended the instrumentation to the target, but did not make the original target's functionality available as a general subroutine. Our unique trampoline design is crucial for extending existing binary software.

In addition to basic detour functionality, Detours also includes functions to edit the DLL import table of any binary, to attach arbitrary data segments to existing binaries, and to inject a DLL into either a new or an existing process. Once injected into a process, the instrumentation DLL can detour any Win32 function, whether in the application or the system libraries.

The following section describes how Detours works. Section 0 outlines the usage of the Detours library. Section 4 describes alternative function-interception techniques and presents a micro-benchmark evaluation of Detours. Section 5 details the usage of Detours to produce distributed applications from local applications, to quantify DCOM overheads, to create a thunking layer for a new COM-based Win32 API, and to implement first chance exception handling. We compare Detours with related work in Section 6 and summarize our contributions in Section 7.

2. Implementation

Detours provides three important sets of functionality: the ability to intercept arbitrary Win32 binary functions on *x86* machines, the ability to edit the import tables of binary files, and the ability to attach arbitrary data segments to binary files. We will describe the implementation of each of these functionalities.

2.1. Interception of Binary Functions

The Detours library facilitates the interception of function calls. Interception code is applied dynamically at runtime. Detours replaces the first few instructions of the *target function* with an unconditional jump to the user-provided *detour function*. Instructions from the target function are preserved in a *trampoline function*. The trampoline consists of the instructions removed from the target function and an unconditional branch to the remainder of the target function.

When execution reaches the target function, control jumps directly to the user-supplied detour function. The detour function performs whatever interception *preprocessing* is appropriate. The detour function can return control to the *source* function or it can call the trampoline function, which invokes the target function without interception. When the target function completes, it returns control to the detour function. The detour function performs appropriate *postprocessing* and returns control to the source function. Figure 1 shows the logical flow of control for function invocation with and without interception.

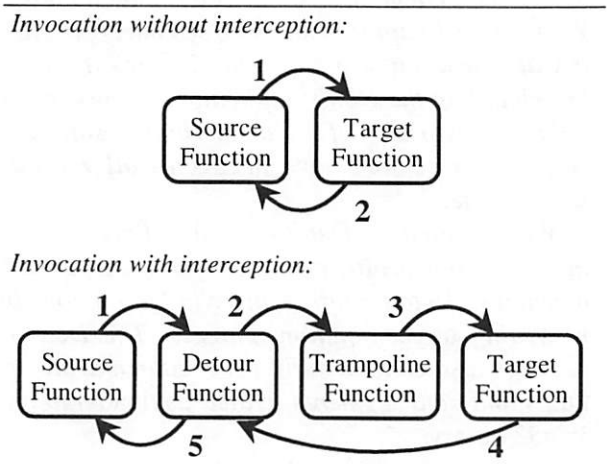


Figure 1. Invocation with and without interception.

The Detours library intercepts target functions by rewriting their in-process binary image. For each target function, Detours actual rewrites two functions: the target function and the matching trampoline function. The trampoline function can be allocated either dynamically or statically. A statically allocated trampoline always invokes the target function without the detour. Prior to insertion of a detour, the static trampoline contains a single jump to the target. After insertion, the trampoline contains the initial instructions from the target function and a jump to the remainder of the target function.

Statically allocated trampolines are extremely useful for instrumentation programmers. For example, in Coign [7], invoking the `Coign_CoCreateInstance` trampoline is equivalent to

invoking the original `CoCreateInstance` function without instrumentation. Coign internal functions can call `Count_CoCreate-Instance` at any time to create a new component instance without concern for whether or not the original function has been rerouted with a detour.

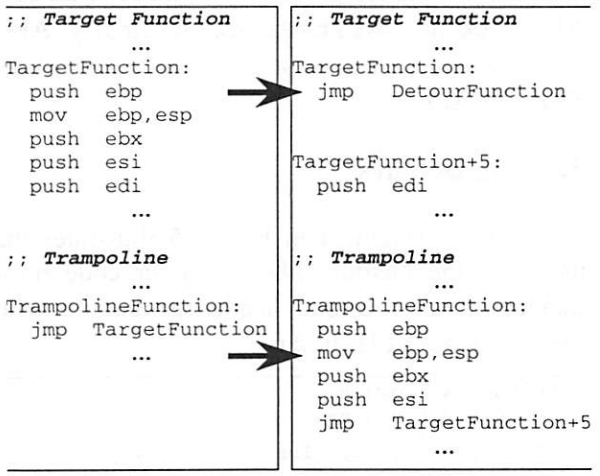


Figure 2. Trampoline and target functions, before and after insertion of the detour (left and right).

Figure 2 shows the insertion of a detour. To detour a target function, Detours first allocates memory for the dynamic trampoline function (if no static trampoline is provided) and then enables write access to both the target and the trampoline. Starting with the first instruction, Detours copies instructions from the target to the trampoline until at least 5 bytes have been copied (enough for an unconditional jump instruction). If the target function is fewer than 5 bytes, Detours aborts and returns an error code. To copy instructions, Detours uses a simple table-driven disassembler. Detours adds a jump instruction from the end of the trampoline to the first non-copied instruction of the target function. Detours writes an unconditional jump instruction to the detour function as the first instruction of the target function. To finish, Detours restores the original page permissions on both the target and trampoline functions and flushes the CPU instruction cache with a call to `Flush-InstructionCache`.

2.2. Payloads and DLL Import Editing

While a number of tools exist for editing binary files [10, 12, 13, 17], most systems research doesn't require such heavy-handed access to binary files. Instead, it is often sufficient to add an extra DLL or data segment to an application or system binary file. In addition to detour functions, the Detours library also contains fully reversible support for attaching arbitrary data segments, called *payloads*, to Win32 binary files and for editing DLL import tables.

Figure 3 shows the basic structure of a Win32 Portable Executable (PE) binary file. The PE format for Win32 binaries is an extension of COFF (the Common Object File Format). A Win32 binary consists of a DOS compatible header, a PE header, a text section containing program code, a data section containing initialized data, an import table listing any imported DLLS and functions, an export table listing functions exported by the code, and debug symbols. With the exception of the two headers, each of the other sections of the file is optional and may not exist in a given binary.

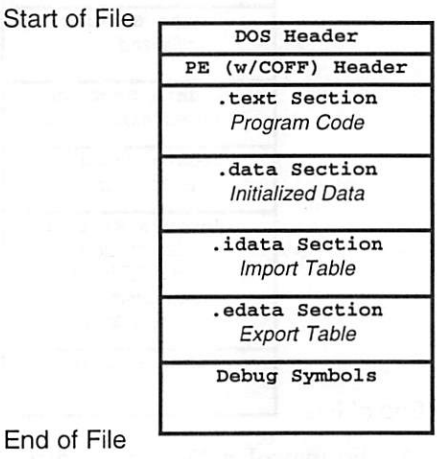


Figure 3. Format of a Win32 PE binary file.

To modify a Win32 binary, Detours creates a new `.detours` section between the export table and the debug symbols. Note that debug symbols must always reside last in a Win32 binary. The new section contains a detours header record and a copy of the original PE header. If modifying the import table, Detours creates the new import table, appends it to the copied PE header, then modifies the original PE header to point to the

new import table. Finally, Detours writes any user payloads at the end of the `.detours` section and appends the debug symbols to finish the file. Detours can reverse modifications to the Win32 binary by restoring the original PE header from the `.detours` section and removing the `.detours` section. Figure 4 shows the format of a Detours-modified Win32 binary.

Creating a new import table serves two purposes. First, it preserves the original import table in case the programmer needs to reverse all modifications to the Win32 file. Second, the new import table can contain renamed import DLLs and functions or entirely new DLLs and functions. For example, Coign [7] uses Detours to insert an initial entry for `coignrte.dll` into each instrumented application. As the first entry in the applications import table, `coignrte.dll` always is the first DLL to run in the application's address space.

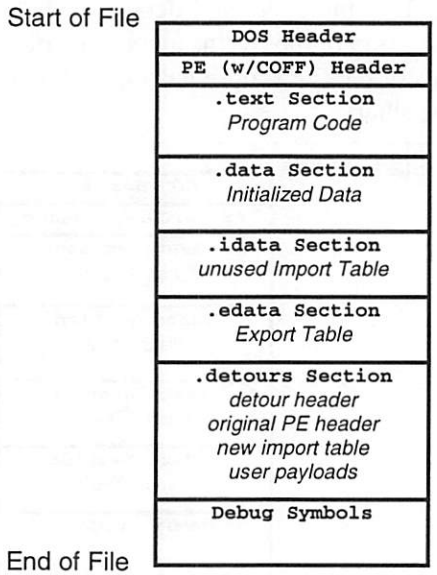


Figure 4. Format of a Detours-modified binary file.

Detours provides functions for editing import tables, adding payloads, enumerating payloads, removing payloads, and rebinding binary files. Detours also provides routines for enumerating the binary files mapped into an address space and locating payloads within those mapped binaries. Each payload is identified by a 128-bit globally unique identifier (GUID). Coign uses Detours to attach per-application configuration data to application binaries.

In cases where instrumentation need be inserted into an application without modifying binary files, Detours provides functions to inject a DLL into either a new or an existing process. To inject a DLL, Detours writes a `LoadLibrary` call into the target process with the `VirtualAllocEx` and `WriteProcessMemory` APIs then invokes the call with the `CreateRemoteThread` API.

3. Using Detours

The code fragment in Figure 5 illustrates the usage of the Detours library. User code must include the `detours.h` header file and link with the `detours.lib` library.

```
#include <windows.h>
#include <detours.h>

VOID (*DynamicTrampoline)(VOID) = NULL;

DETOUR_TRAMPOLINE(
  VOID WINAPI SleepTrampoline(DWORD),
  Sleep
);

VOID WINAPI SleepDetour(DWORD dw)
{
  return SleepTrampoline(dw);
}

VOID DynamicDetour(VOID)
{
  return DynamicTrampoline();
}

void main(void)
{
  VOID (*DynamicTarget)(VOID) = SomeFunction;

  DynamicTrampoline
    =(FUNCPTR)DetourFunction(
       (PBYTE)DynamicTarget,
       (PBYTE)DynamicDetour);

  DetourFunctionWithTrampoline(
    (PBYTE)SleepTrampoline,
    (PBYTE)SleepDetour);

  // Execute the remainder of program.

  DetourRemoveTrampoline(SleepTrampoline);
  DetourRemoveTrampoline(DynamicTrampoline);
}
```

Figure 5. Sample Instrumentation Program.

Trampolines may be created either statically or dynamically. To intercept a target function with a static trampoline, the application must create the trampoline with the `DETOUR_TRAMPOLINE`

macro. DETOUR_TRAMPOLINE takes two arguments: the prototype for the static trampoline and the name of the target function.

Note that for proper interception the prototype, target, trampoline, and detour functions must all have exactly the same call signature including number of arguments and calling convention. It is the responsibility of the detour function to copy arguments when invoking the target function through the trampoline. This is intuitive as the target function is just a subroutine callable by the detour function.

Using the same calling convention insures that registers will be properly preserved and that the stack will be properly aligned between detour and target functions.

Interception of the target function is enabled by invoking the DetourFunctionWithTrampoline function with two arguments: the trampoline and the pointer to the detour function. The target function is not given as an argument because it is already encoded in the trampoline.

A dynamic trampoline is created by calling DetourFunction with two arguments: a pointer to the target function and a pointer to the detour function. DetourFunction allocates a new trampoline and inserts the appropriate interception code in the target function.

Static trampolines are extremely easy to use when the target function is available as a link symbol. When the target function is not available for linking, a dynamic trampoline can be used. Often a function pointer to the target function can be acquired from a second function. For those times, when a pointer to the target function is not readily available, DetourFindFunction can find the pointer to a function when it is either exported from a known DLL, or if debugging symbols are available for the target function's binary[1].

DetourFindFunction accepts two arguments, the name of the binary and the name of the function. DetourFindFunction returns either a valid pointer to the function or NULL if the symbol for the function could not be found. DetourFindFunction first attempts to locate the function using the Win32 LoadLibrary and GetProcAddress APIs. If the function is not found in the export table of the DLL, DetourFindFunction uses the ImageHlp library to search available debugging symbols. The function pointer returned by DetourFindFunction can be given to DetourFunction to create a dynamic trampoline.

Interception of a target function can be removed by invoking the DetourRemoveTrampoline function.

Note that because the functions in the Detours library modify code in the application address space, it is the programmer's responsibility to ensure that no other threads are executing in the address space while a detour is inserted or removed. An easy way to insure single-threaded execution is to call functions in the Detours library from a DllMain routine.

4. Evaluation

Several alternative techniques exist for intercepting function calls. Alternative interception techniques include:

Call replacement in application source code. Calls to the target function are replaced with calls to the detour function by modifying application source code. The major drawback of this technique is that it requires access to source code.

Call replacement in application binary code. Calls to the target function are replaced with calls to the detour function by modifying application binaries. While this technique does not require source code, replacement in the application binary does require the ability to identify all applicable call sites. This requires substantial symbolic information that is not generally available for binary software.

DLL redirection. If the target function resides in a DLL, the DLL import entries in the binary can be modified to point to a detour DLL. Redirection to the detour DLL can be achieved by either replacing the name of the original DLL in the import table before load time or replacing the function addresses in the indirect import jump table after load [2]. Unfortunately, redirecting to the detour DLL through the import table fails to intercept DLL internal calls and calls on pointers obtained from the LoadLibrary and

[1] Microsoft ships debugging symbols for the entire Windows NT operation system as part of the retail release. These symbols can be found in the \support\symbols directory on the OS distribution media.

`GetProcAddress` APIs early in an applications execution.

Breakpoint trapping. Rather than replace the DLL, the target function can be intercepted by inserting a debugging breakpoint into the target function. The debugging exception handler can then invoke the detour function. The major drawback to breakpoint trapping is that debugging exceptions suspend all application threads. In addition, the debug exception must be caught in a second operating-system process. Interception via break-point trapping has a high performance penalty.

Table 1 lists times for intercepting either an empty function or the `CoCreateInstance` API. Times are on a 200 MHz Pentium Pro. Rows list the time to invoke the functions without interception, with interception through call replacement, with interception through DLL redirection, with interception using the Detours library, or with interception through breakpoint trapping. As can be seen, function interception with Detours library has only minimal overhead (less than 400 ns in either case).

Interception Technique	Intercepted Function	
	Empty Function	`CoCreate-Instance`
Direct	0.113µs	14.836µs
Call Replacement	0.143µs	15.193µs
DLL Redirection	0.143µs	15.193µs
Detours Library	0.145µs	15.194µs
Breakpoint Trap	229.564µs	265.851µs

Table 1. Comparison of Interception Techniques.

5. Experience

The Detours package has been used extensively in Microsoft Research over the last two years to instrument and extend Win32 applications and the Windows NT operating system.

Detours was originally developed for the Coign Automatic Distributed Partition System [7]. Coign converts local desktop applications built from COM components into distributed client-server applications. During profiling, Coign uses Detours to intercept calls to COM instantiation functions such as `CoCreateInstance`. The detour functions invoke the original library functions through trampolines, then wrap output interface pointers in an additional instrumentation layer (for more details see [8]). The instrumentation layer measures inter-component communication to determine how application components should be partitioned across a network. During distributed executions, new Coign detour functions intercept calls to COM instantiation functions and re-route those calls to distributed machines. In essence, Coign extends the COM library to support intelligent remote invocation. Whereas DCOM supports remote invocation of a few COM instantiation functions, Coign supports remote invocation for approximately 50 COM functions through detour extensions. Coign uses Detours' DLL redirection functions to attach a runtime loader and the payload functions to attach profiling data to application binaries.

Our colleagues have used Detours to instrument the user-mode portion of the DCOM protocol stack including marshaling proxies, DCOM runtime, RPC runtime, WinSock runtime, and marshaling stubs [11]. The resultant detailed analysis was then used to drive a re-architecture of DCOM for fast user-mode networks. While they could have used source code modifications to produce a special profiling version of DCOM, the source-based instrumentation would have been version dependent and shared by all DCOM applications on the profiling machine. With binary instrumentation based on Detours, the profiling tool can be attached to any Windows NT 4 build of DCOM and only effects the process being profiled.

In another extension exercise, Detours was used to create a thunking layer for COP (the Component-based Operating System Proxy) [14]. COP is a COM-based version of the Win32 API. COP aware applications access operating system functionality through COM interfaces, such as `IWin32FileHandle`. Because the COP interfaces are distributable with DCOM, a COP application can use OS resources, including file systems, keyboards, mice, displays, registries, etc., from any machine in a network. To provide support for legacy applications, COP uses detour functions to intercept all application calls to the Win32 APIs. Native application API calls are converted to calls on COP interfaces. At the bottom, the COP implementation communicates

with the underlying operating system through trampoline functions. COP requires no modifications to application binaries. At load time, the COP DLL is injected into the application's address space with Detours' injection functions. Through its simple interception, Detours has facilitated this massive extension of the Win32 API.

Finally, to support Software Distributed Shared Memory (SDSM) systems, we have implemented a first chance exception filter for Win32 structured exception handling. The Win32 API contains an API, `SetUnhandledExceptionFilter`, through which an application can specify an exception filter to execute should no other filter handle an application exception. For applications such as SDSM systems, the programmer would like to insert a first-chance exception filter to remove page faults caused by the SDSM's manipulation of VM page permissions. Windows NT does not provide such a first-chance exception filter mechanism. A simple detour intercepts the exception entry point from kernel mode to user mode (`KiUserExceptionDispatcher`). With only a few lines of code, the detour function calls a user-provided first-chance exception filter and then forwards the exception, if unhandled, to the default exception mechanism through a trampoline.

6. Related Work

Detours are an extension of the general technique of code patching. To intercept execution, an unconditional branch or jump is inserted into the desired point of interception in the target function. Code overwritten by the unconditional branch is moved to a code patch. The code patch consists of either the instrumentation code or a call to the instrumentation code followed by the instructions moved to insert the unconditional branch and a jump to the first instruction in the target function after the unconditional branch. Logically, a code patch can be prepended to the beginning of a function, inserted at some arbitrary point in a function, or appended to the end of a function.

Whereas a code patch invokes instrumentation then continues the target function, our technique transfers control completely to the detour function which can invoke the original target function through the trampoline at its leisure. The trampoline gives instrumentation complete freedom to invoke the semantics of the original function as a callable subroutine at any time.

Techniques for code patching have existed since the dawn of digital computing [3-5, 9, 15]. Code patching has been applied to insert debugging or profiling code. In the distant past, code patching was generally considered to be a much more practical update method than re-compiling the entire application. In addition to debugging and profiling, Detours has also been used to resourcefully extend the functionality of existing systems [7, 14].

While recent systems have extended code patching to parallel applications [1] and system kernels [16], Detours is to our knowledge the only code patching system that preserves the semantics of the target function as a callable subroutine. The detour function replaces the target function, but can invoke its functionality at any point through the trampoline. Our unique trampoline design makes it trivial to extend the functionality of existing binary functions.

Recent research has produced a class of detailed binary rewriting tools including Atom [13], Etch [12], EEL [10], and Morph [17]. In general, these tools take as input an application binary and an instrumentation script. The instrumentation script passes over the binary inserting code between instructions, basic blocks, or functions. The output of the script is a new, instrumented binary. In a departure for earlier systems, DyninstAPI [6] can modify applications dynamically.

Detours' primary advantage over detailed binary rewriters is its size. Detours adds less than 18KB to an instrumentation package whereas detailed binary rewriters add at least a few hundred KB. The cost of Detours small size is an inability to insert code between instructions or basic blocks. Detailed binary rewriters can insert instrumentation around any instruction through sophisticated features such as free register discovery. Detours relies on adherence to calling conventions in order to preserve register values. While detailed binary rewriters support insertion of code before or after any basic instruction unit, they do not preserve the semantics of the

uninstrumented target function as a callable subroutine.

7. Conclusions

The Detours library provides an import set of tools to the arsenal of the systems researcher. Detour functions are fast, flexible, and friendly. A detour of `CoCreateInstance` function has less than a 3% overhead, which is an order of magnitude smaller than the penalty for breakpoint trapping. The Detours library is very small. The runtime consists of less than 40KB of compiled code although typically less than 18KB of code is added to the users instrumentation.

We are currently working on versions of Detours for Windows 98 and the Alpha processors. The Alpha port should be trivial due to the uniform size of instructions in the Alpha's RISC architecture.

Unlike DLL redirection, the Detours library intercepts both statically and dynamically bound invocations. Finally, the Detours library is much more flexible than DLL redirection or application code modification. Interception of any function can be selectively enabled or disabled for each process individually at execution time.

Our unique trampoline preserves the semantics of the original, uninstrumented target function for use as a subroutine of the detour function. Using detour functions and trampolines, it is trivial to produce compelling system extensions without access to system source code and without recompiling the underlying binary files. Detours makes possible a whole new generation of innovative systems research on the Windows NT platform.

Availability

The Detours library is freely available for research purposes. It can be found in either source form or as a compiled library at `http://research.microsoft.com/sn/detours`.

References

[1] Aral, Ziya, Illya Gertner, and Greg Schaffer. Efficient Debugging Primitives for Multiprocessors. *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 87-95. Boston, MA, April 1989.

[2] Balzer, Bob. Mediating Connectors, Release 2.0. University of Southern California, Information Sciences Institute, Los Angeles, CA, April 1999.

[3] Digital Equipment Corporation. *DDT Reference Manual*, 1972.

[4] Evans, Thomas G. and D. Lucille Darley. DEBUG -- An Extension to Current Online Debugging Techniques. *Communications of the ACM*, 8(5), pp. 321-326, May 1965.

[5] Gill, S. The Diagnosis of Mistakes in Programmes on the EDSAC. *Proceedings of the Royal Society, Series A*, 206, pp. 538-554, May 1951.

[6] Hollingsworth, Jeffrey K. and Bryan Buck. DyninstAPI Programmer's Guide, Release 1.2. Computer Science Department, University of Maryland, College Park, MD, September 1998.

[7] Hunt, Galen C. and Michael L. Scott. The Coign Automatic Distributed Partitioning System. *Proceedings of the Third Symposium on Operating System Design and Implementation (OSDI '99)*, pp. 187-200. New Orleans, LA, February 1999. USENIX.

[8] Hunt, Galen C. and Michael L. Scott. Intercepting and Instrumenting COM Applications. *Proceedings of the Fifth Conference on Object-Oriented Technologies and Systems (COOTS'99)*, pp. *to Appear.* San Diego, CA, May 1999. USENIX.

[9] Kessler, Peter. Fast Breakpoints: Design and Implementation. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pp. 78-84. White Plains, NY, June 1990.

[10] Larus, James R. and Eric Schnarr. EEL: Machine-Independent Executable Editing. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 291-300. La Jolla, CA, June 1995.

[11] Li, Li, Alessandro Forin, Galen Hunt, and Yi-Min Wang. High-Performance Distributed Objects over a System Area Network. *Proceedings of the Third USENIX NT Symposium*. Seattle, WA, July 1999.

[12] Romer, Ted, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and J. Bradley Chen. Instrumentation and Optimization of Win32/Intel Executables Using Etch. *Proceedings of the USENIX Windows NT Workshop 1997*, pp. 1-7. Seattle, WA, August 1997. USENIX.

[13] Srivastava, Amitabh and Alan Eustace. ATOM: A System for Building Customized Program Analysis Tools. *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pp. 196-205. Orlando, FL, June 1994.

[14] Stets, Robert J., Galen C. Hunt, and Michael L. Scott. Component-based Operating System APIs: A Versioning and Distributed Resource Solution. *IEEE Computer*, 32(7), July 1999.

[15] Stockham, T.G. and J.B. Dennis. FLIT- Flexowriter Interrogation Tape: A Symbolic Utility Program for the TX-0. Department of Electical Engineering, MIT, Cambridge, MA, Memo 5001-23, July 1960.

[16] Tamches, Ariel and Barton P. Miller. Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels. *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI '99)*, pp. 117-130. New Orleans, LA, February 1999. USENIX.

[17] Zhang, Xiaolan, Zheng Wang, Nicholas Gloy, J. Bradley Chen, and Michael D. Smith. System Support for Automated Profiling and Optimization. *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*. Saint-Malo, France, October 1997.

Evaluating Windows NT Terminal Server Performance

Alexander Ya-li Wong, Margo I. Seltzer
Harvard University, Division of Engineering and Applied Sciences
{aywong,margo}@eecs.harvard.edu

Abstract

With the introduction of Windows NT, Terminal Server Edition (TSE), Microsoft finally brings to Windows the "thin-client" computing model the X Window System has offered Unix for a decade. TSE's two most salient features are the provision of multi-user login service and the provision of that service remotely, over a network link. These features distinguish TSE from previous Microsoft operating systems not only by functionality, but also by how its performance ought to be measured. Because TSE's primary service is interactive, user-perceived latency is more important than ever. In this paper, we examine the resource consumption and latency characteristics of shared usage on a TSE system. We find that the introduction of remote, multi-user access has added to the minimal level of resource consumption, that the efficiency of TSE's RDP protocol is generally good but degrades on dynamic user interface elements, and that TSE can exhibit poor latency performance when subjected to high processor, memory, and network load.

1 Introduction

With Microsoft's introduction of Windows NT, Terminal Server Edition (TSE), the Windows platform has acquired the multi-user, remote access capabilities that have been available for Unix since X-Windows appeared a decade ago. Microsoft seems to have awoken to the possibility that "thin-client" computing is a viable and even desirable alternative to their vision of "Windows on Every Desktop". Report after industry report has revealed a thirst among IT managers for the lower total cost of ownership offered by thin-client computing. Microsoft hopes to quench that thirst with TSE, which they bill as a way to extend the life of older hardware, bring Windows applications to non-Windows desktops, and rein in per-user management costs [11, 12].

1.1 Inside TSE

The TSE environment is composed of three major components [13]:

Terminal Server - The TSE core is a fork of the NT 4.0 Server codebase diverging sometime after the release build of 4.0 Server. Key parts of the OS were modified to support multiple concurrent users. Kernel objects are virtualized across user sessions, the Virtual Memory Manager supports per-session mappings of kernel address space, and the kernel supports code page sharing across sessions. The "Terminal Service" was introduced to manage user sessions.

Remote Display Protocol (RDP) - RDP runs over TCP/IP and makes user sessions available remotely. It includes control messages (e.g., keystrokes, mouse events) sent from the clients to the server and display messages sent from the server to the clients. Server-side RDP support includes per-user replacements for display, keyboard, and mouse drivers, which bind to an RDP stack instead of local hardware devices to enable remote display and control.

Terminal Clients - Terminal clients communicate with the Terminal Server using RDP. They display graphics delivered from the server on local video hardware and capture local keyboard and mouse input for delivery back to the server. Clients are available for Windows for Workgroups, CE, 95, 98, NT 3.51, and NT 4.0.

1.2 Evaluating TSE

Along with the new functionality offered by TSE come new requirements for measuring its performance. As a multi-user, interactive, remote access operating system, TSE cannot be evaluated with the same metrics used for either its predecessors or for other operating systems.

For example, it is not sufficient to assign a simple Winstone, SYSmark, or SPECmark value to a machine running TSE. Ultimately, those interested in deploying TSE need to know the maximum number of concurrent users their servers can support, and they need benchmark results to be expressed in these terms. Moreover, the particular characteristics of the TSE environment make it incompatible with most existing benchmarks. TSE is:

Multi-User - TSE is designed to support heavy, concurrent use on today's hardware [1]. Benchmarks designed for single-user operating systems are not appropriate be-

cause a single user multi-tasking is not equivalent to multiple users uni-tasking or multi-tasking. On single-user systems, although asynchronous background tasks may consume system resources, system load is still typically limited only by the rate at which the human user interacts with the foreground application. Furthermore, on a multi-user system there can be many foreground applications (one for each user), so latency demands must be met by more than just a single process.

Interactive - Unlike HTTP or database servers for which throughput is critical, TSE's primary service is interactive login. As argued by Endo et. al, latency, not throughput, is the paramount performance criterion for this type of system [5]. Any useful metric should yield information necessary to gauge whether the system satisfies the latency demands of users.

Remote Access - On single-user systems like Windows 98 and NT Workstation, user interface richness and sophistication consume and are constrained by locally available video subsystem bandwidth. In the TSE environment, the video subsystem at the server is irrelevant and the GUI is instead constrained by network bandwidth, the efficiency of RDP, and the video hardware at the terminal.

This paper analyzes the resource demands of TSE's multi-user and remote display capabilities. In particular, we evaluate resource consumption of processor cycles, memory, and network bandwidth. Then, we examine the effect of load on TSE's ability to maintain low latency to yield a smooth user experience.

In Section 2, we describe our experimental environment. In Section 3, we consider resource consumption of the processor, memory, and network. In Section 4, we evaluate TSE's latency characteristics under load. In Section 5, we review related work, and we conclude in Section 6.

2 Experimental Environment

Our testbed consisted of a server, a client, and a network listening host connected to a NetGear EN104TP 10Mbps Ethernet hub. The server was a Celeron-333, Intel 440EX AGPset, 48MB SDRAM, 4GB IDE IBM DCAA-34330, and a NetGear FA-310 NIC. The client was a Pentium II-400, Intel 440BX AGPset, 128MB SDRAM, 11GB IDE Maxtor 91152D8, and a 3Com 3C905B NIC. The listening host was a Pentium-233, Intel 440TX PCIset, 96MB EDO RAM, 2GB IDE IBM DTNA-22160, and a 3Com 3C589C NIC. The server ran TSE build 419, the client ran Windows 98 with TSE client build 419, and the listening host ran the 2.0.36 Linux kernel. All NIC's were set to 10Mbps operation.

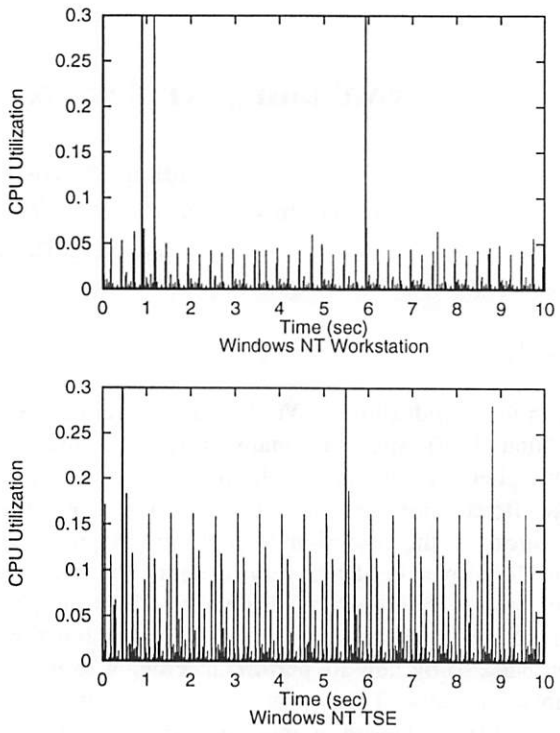


Figure 1: A comparison of the idle-state processor activity in Windows NT Workstation 4.0 and TSE.

3 Resource Consumption

In a server operating system, resource sharing is typically defined in terms of concurrent processes, transactions, or queries. For TSE, the sensible unit for quantifying resource consumption is the user. In this section, we evaluate TSE's scalability by examining per-user consumption of server-side processor cycles, memory, and network bandwidth. Resource consumption is highly dependent on the particular applications being used, so we provide consumption data as generically as possible to allow extrapolation to scenario-dependent usage.

3.1 Processor

The changes made to the core of NT to enable multi-user, remote access would likely increase processor overhead, which can translate to user-perceptible latency even under no server load.

Endo et. al introduced a technique for measuring user-perceived latency they call "measuring lost time" [5]. Using the Pentium Performance Counters and system idle loop instrumentation, they are able to determine when, and for how long, the CPU is busy handling user input events. This technique can also be used to generate detailed visualizations of CPU utilization.

Their paper reported idle-state CPU activity profiles for Windows 95, NT 3.51, and NT 4.0. To quantify the baseline activity introduced in the transition from NT to TSE, we reproduce their experiment for NT 4.0 and TSE. Figure 1 shows that TSE is indeed more active than NT when idle. The difference is due primarily to a large number of events in TSE that last between 200 and 400ms. This increased activity can probably be attributed to the additional services necessary to support multi-user, remote access. Viewing the latencies over time as a cumulative distribution reveals that TSE has 71.0% more idle-state activity than NT.

3.2 Memory

In a single-user operating system, the user runs a number of applications, each of which consumes memory for the executable image and the dynamic stack and heap. In a multi-user system, many users may share the same applications. Code page sharing, which is supported in TSE, reduces total memory usage by backing only a single copy of the code text and mapping it read-only into the address space of each application instance [13].

Therefore, memory is consumed by applications in TSE in two ways. First, each new instance of an application consumes at least as much memory as the size of its stack and heap. Second, the greater the number of distinct applications used, the greater the combined memory usage for all of the applications' shared code pages.

3.2.1 User Sessions

We used Microsoft Process Viewer to collect memory consumption data on TSE and a number of popular Windows applications.

Process	Cold	Warm	Disconnect
csrss.exe	360	452	452
explorer.exe	732	1,368	1,368
loadwc.exe	424	424	424
nddeagnt.exe	300	300	300
winlogin.exe	396	700	700
Total	2,212	3,244	3,244

This table lists the processes automatically started with each unique user login session and the amount, in kilobytes, of private, committed memory in their address spaces. "Private" means that the memory is not mapped and is visible only to the local process. "Committal" is the Microsoft term for the allocation of backing store to virtual memory pages. Memory may be "reserved" without committal, and reserved memory may be released without ever having been committed [16]. Throughout our discussion, we conservatively include only private, committed memory in our definition of memory utilization. We do not include amortized shared code page costs for executable text, nor mapped, committed memory that may or may not be shared.

The "Cold" column reports values for a login with no subsequent user activity. The total for the Cold column is therefore a lower bound on per-user memory usage because each new user login will always consume *at least* this much memory. The "Warm" column reports values after modest usage. We see that the stack and heap size for the `csrss` (client-server runtime subsystem) and `explorer` (the shell) processes increases somewhat, while it does not for the other processes. The "Disconnect" column reports memory usage when the user has disconnected from the session (which is not a full logoff and allows a later reconnect). Clearly, memory is not explicitly conserved upon disconnect. We assume that TSE relies on the pagefile for this.

We should note that TSE clients have the option of dispensing with the default shell, the Explorer, and running just a specific application in a user session. This admits the possibility of using a lighter alternative such as the DOS Prompt (command.com), which at 224KB of cold, private, committed memory, is less than one-third the size of the Explorer.

3.2.2 Applications

This table presents memory usage statistics in kilobytes for a sampling of popular Windows applications. Measurements were taken immediately after the application was launched with the closest approximation of a null document. Actual mileage will vary.

Application	Mapped	Private
Notepad	1,068	148
Word 97	1,796	932
Excel 97	1,788	284
PowerPoint 97	1,816	300
Access 97	2,324	1,176
Outlook '97	6,532	1,356
IE 4.0	1,368	800
Netscape 4.08	1,460	1,396
WordPerfect 8	1,276	1,212
Visual C++ 5.0	1,046	1,316

All memory reported is committed. The first column reports the amount of memory that is mapped and is potentially shared. This provides an estimate for the amount of shareable code text in the application that will not be duplicated for each running instance. The second column reports the amount of private, committed memory, which is most likely the stack and heap space. This gives a conservative lower-bound for the per-user memory utilization of each application. These values surely

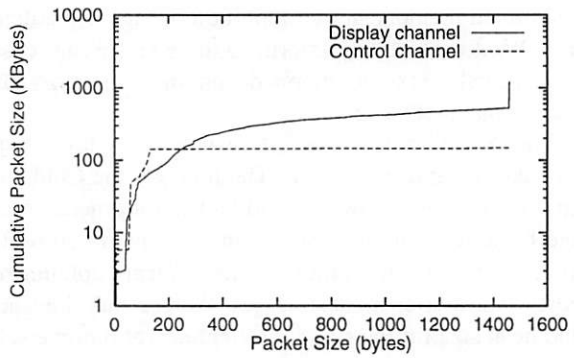


Figure 2: RDP control and display channel packet size distribution. Note the log-scaling.

rise as users open documents and otherwise begin to actually use the applications. Exactly how much depends highly on the application and the documents.

3.3 Network

Network load generated by TSE is a function of the user interface richness and complexity of the applications in use and the efficiency with which RDP encodes the dynamics of those interfaces.

3.3.1 RDP Efficiency

There is no published specification for RDP. Therefore, we are not yet able to offer detailed explanations for all of our RDP observations. An RDP reverse-engineering effort is part of our ongoing work.

First, we investigated the network packet size distribution of a live RDP message stream. We had a user open a TSE session from *client* to *server* and work for several minutes, composing a document in Word 97, browsing the NT Event Log, and navigating the filesystem using Explorer. Figure 2 shows the cumulative distribution of both control and display channel packets.

The data indicate that the control channel is dominated by packets on the order of 100 bytes. For these small control messages, the overhead imposed by 20 byte IP headers is considerable. In non-routed TSE environments, a scheme like the *x-kernel* virtual-IP (VIP) network stack could reduce overhead by omitting the IP header [9]. The total number of control bytes transferred in this test was 282,258. Without IP headers, only 232,158 would be transferred, yielding a 17.6% savings in overhead.

Unfortunately, the benefit of such an optimization is partially mitigated by traffic on the display channel where the average packet size is much larger. Accounting for both channels, a VIP-like optimization would reduce total bytes transferred from 1,642,337 to 1,533,797, a savings of only 6.6%. Regardless, a VIP scheme could still measurably improve latency performance. Because display messages are typically sent only in response to control messages, reducing the average packet size on just the control channel can reduce overall latency.

3.3.2 Decompositional Analysis

In the previous experiment, we showed that the packet size distribution of a "typical" TSE session was amenable to optimization. However, the efficacy of the optimization depends greatly on the distribution. A user running a different mix of applications could produce a dramatically different distribution.

This illustrates the importance of considering user behavior and application mix in the TSE world. A common attribute of previous papers on TSE performance is a definition of what the authors believe to be "typical" user behavior. Because these definitions of behavior strongly influence resource utilization results, it is important to analyze their validity and realism.

Therefore, we performed decompositional analysis on RDP in terms of smaller, more discrete operations. This helped us to gain some understanding of how RDP works from a microscopic view. All values reported here are in bytes per second. For comparison, keep in mind that 10Mbps Ethernet has a bandwidth of 1,250 bytes per second.

Keystrokes and Typing

Trace data from *tcpdump* reveals that a single keystroke made at a TSE client generates 2 RDP messages on the control channel:

```
client.1972 > server.3389: P 708:768(60)
client.1972 > server.3389: P 768:828(60)
```

These two server-bound messages are presumably *KeyDown* and *KeyUp*, and each appears to be 60 bytes. Note that 60 is the size of the TCP/IP packet's data payload. In reality, delivery of a 60 byte RDP message requires link-layer transmission of 118 bytes since each message is wrapped with 40 bytes of TCP/IP headers and 18 bytes of Ethernet framing. A single keystroke, then, generates 232 bytes of traffic on the control channel.

Using this information, we can derive the theoretical control channel bandwidth utilization of typing. If we assume 5.29 characters per word (as in the present work), then for every 10wpm a typist is fast, he will generate

$$\frac{10 \text{ words}}{\text{min}} \cdot \frac{5.29 \text{ chars}}{\text{word}} \cdot \frac{232 \text{ bytes}}{\text{char}} \cdot \frac{1 \text{ min}}{60 \text{ sec}} = 0.20\text{KBps}$$

of network traffic. A 75wpm typist therefore generates 1.51KBps of load.

This theoretical value is borne out by empirical observation. In our test, a user typed at approximately 75wpm into the Notepad application. Bandwidth was measured on the control channel only and was found to be 1.56KBps.

Unfortunately, we cannot apply this same generalizing analysis to derive display channel utilization because the size and number of display messages sent in response to keystrokes varies from application to application. We did, however, measure overall bandwidth utilization for "typical" typing behavior in two Windows word processors, Word 97 and WordPerfect 8. Both word processors were set to 10pt Times Roman font and 100% magnification page layout view. For a user typing at 100wpm, Word generated 6.26KBps of display traffic and WordPerfect generated 11.66KBps.

Typing under RDP is not particularly bandwidth-intensive. On a 10 Mbps Ethernet, one should be able to support on the order of 100 users composing simple text documents in Word or WordPerfect.

Mouse Movement

If the cursor is handled at the client, we would expect mouse movement to generate little or no RDP traffic. But, because the Win32 GUI can change cursors depending on location, the server-side application logic must be constantly informed by the client of the cursor's location. However, the server sends no display messages in response to mouse movement, except in the case of a cursor change. The TSE client software presumably handles the overlay of the cursor image over the display data received from the server.

We observe that continuous movement of the mouse in random directions on an open area of the desktop (to avoid cursor changes) generates 1.95KBps of network traffic. The simple behavior of moving the mouse appears unlikely to produce bandwidth utilization above perhaps 2KBps.

Menu Navigation

Another typical operation performed by Windows users is menu navigation. To measure the bandwidth consumption of this operation, we performed a simple human-driven depth-first traversal of the "Start Button" menu tree on our system. This activity generated a network load of 3.12KBps. Note that this value includes the 1.95KBps of traffic generated by mouse movement alone. This means that approximately 1.17KBps of traffic was generated by the movement of the menu highlight and the periodic display of newly activated cascade menus.

We also performed a similar human-driven test in Word and WordPerfect. In this test, the user hit the 'Alt' key to select the menu bar, and then held down the right arrow key, causing each top-level menu to be displayed briefly. WordPerfect generated 17.13KBps and Word 39.82KBps. We attribute this difference to the fact that Word's menus are larger and include graphical icons. Although menu navigation behavior is rarely sustained, 40KBps is a realistic estimate of burst rate.

Scrolling

Scrolling through a document is another common operation. Although the display message behavior of scrolling is highly application-dependent, we can measure bandwidth utilization in a realistic usage scenario.

The test we performed had a user scroll though the same multi-page document in Word and WordPerfect by holding down the *PgDn* key. The document view was set to Page Layout at 100% magnification. The body of the document consisted almost entirely of 10pt Times Roman text, interrupted periodically by a small table.

The average bandwidth over the duration of the scroll operation was 59.24KBps for Word and 192.54KBps for WordPerfect. Word seems to render rapid text movement more efficiently, at least with respect to the primitives used in RDP. Although scrolling is an intermittent operation, in a realistic TSE deployment scenario, one can easily imagine seeing prolonged bursts of up to 60KBps in Word and almost 200KBps in WordPerfect as users scroll through long documents.

Summary

Typical use of Windows applications seems to consume modest amounts of bandwidth. Users rarely, if ever, perform any of the above described operations simultaneously. Therefore, the peak bandwidth utilization of a typical user will generally be the maximum utilization among all of the operations. That is, such users will generally produce, at most, 60KBps-200KBps of traffic (from intensive scrolling).

3.3.3 Animations

Of course, user interaction with Windows and Windows applications is not limited to just the operations described above. Moreover, user interfaces have steadily grown more rich and sophisticated over time.

Basic user activity in a non-TSE Windows environment rarely saturates system resources for extended lengths of time, often leaving the CPU idle. Modern Windows programmers have, very reasonably, sought to exploit this idle CPU capacity to enhance the user experience. We identify two relevant such exploits here.

The first is the use of animation. Although the motivation for such enhancements is often intuitive, previous work has experimentally shown that animation can create the illusion of reduced latency through visual conti-

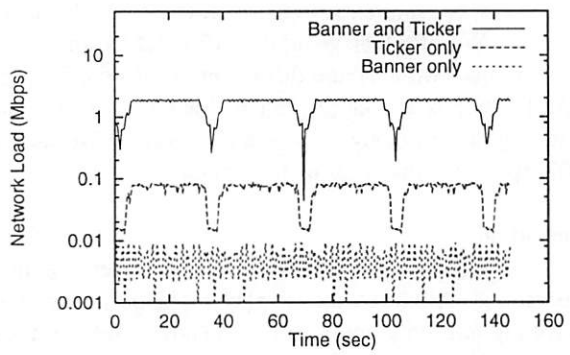


Figure 3: Comparison of network utilization of just the marquee, just the banner, and both at once.

guity [2]. Examples include windows that visually shrink and expand when minimized and maximized, and Office 97/Windows 98 style animated menus that "unfold" or "slide out" upon activation.

A second avenue of idle CPU exploitation is asynchronous multi-threading. Examples include background file copying and spell-checking. Multi-threading has also appeared in user interfaces in the form of asynchronous animations. They can be used to inform users of status, as the fidgety "Office Assistant" characters do for users of Office 97. Often, the animation itself is the application, such as screensavers or animated banner advertisements on webpages.

Animation arguably makes the user experience less static and more engaging, but can exact a high bandwidth cost. The rapid frame rates required for smooth and effective animation can generate a large amount of RDP display traffic. Continuous asynchronous animations also generate "background noise" on the network. So, in the course of exploiting idle CPU and video subsystem resources, Windows developers have created applications whose user interfaces can be potentially ill-behaved in a remote computing environment such as TSE.

Ancillary Animation

First, we quantify the network utilization of some simple interface-enhancing animations. Windows supports blinking cursors and even permits the user to adjust the blink rate in the Control Panel. At the maximum setting, a single blinking cursor in the Notepad application generates a consistent 0.50KBps of network activity. Office 97/Windows 98 style menu animations increase the bandwidth utilization of the previously discussed menu navigation test in Word from 39.82KBps to 48.88KBps. The Office 97 Assistant character ClipIt, even when idle, blinks his eyes periodically. This produces a 0.30KBps level of activity. When he becomes more animated, bandwidth usage also increases. When a user saves a file in Word 97, ClipIt folds his paper-clip body into a box and tosses his eyeballs in, indicating that he is storing something valuable. This animation produces a peak rate of 5.00KBps and lasts for approximately 7 seconds.

Animation as the Application

Fortunately, these levels of additional network traffic are relatively harmless. Other types of animation, however, can be quite costly. In particular, today's web pages are filled with animated GIF advertisements and Java- and HTML-based stock and news tickers.

To study such media-intensive webpages more carefully, we created a synthetic webpage that included one animated 468x60 pixel GIF banner advertisement and a one-line scrolling news ticker. The page was served by Apache on the *listen* host to Internet Explorer running on *server* but being displayed via RDP on *client*.

This type of animated page is not uncommon on today's web, and might even be considered modest. But, as shown by the top bandwidth trace in Figure 3, simply displaying it in Internet Explorer produces a sustained average network load of 1.60Mbps. The plateaus of higher activity average 1.89Mbps. The periodicity of the trace is due to the periodicity of the scrolling news ticker.

Such levels of network activity make multi-user service over standard 10Mbps Ethernet unfeasible. If just 5 users open their browsers to a page like this, the network link is already totally saturated. Although many TSE administrators may, by policy, prohibit the use of web browsers or enforce the deactivation of webpage animations, this remains an important issue to consider when developing a "realistic" user behavior profile.

Managing Animation: The Bitmap Cache

Interestingly, when the two elements of our synthetic webpage are displayed separately, the network load characteristics are markedly different. Figure 3 also shows load traces for displaying the banner advertisement and news ticker each in isolation.

Average bandwidth for the ticker alone is 0.07Mbps, and for the banner alone it is 0.01Mbps. These values do not sum to 1.89Mbps, or even 1.60Mbps, so the network load behavior of RDP is clearly non-linear with respect to the complexity and quantity of this animation.

This behavior implies the presence of a client-side bitmap cache large enough to store all the frames of one animation or the other, but not both. When the two animations are combined, their competing frames overflow the cache and every miss generates a full bitmap transfer over the network. When the frames do fit into the cache, we presume that display data need not be transferred, and only small "swap bitmap" messages are exchanged.

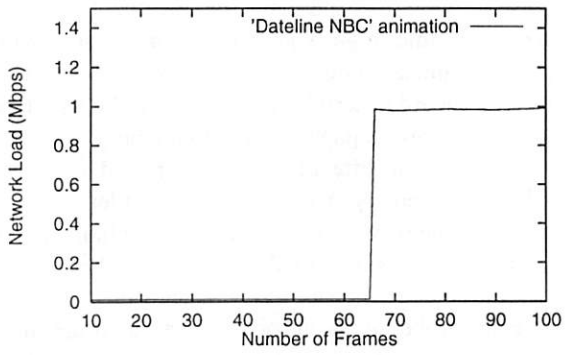


Figure 4: Network load for displaying animations of various frame counts illustrating the size of the bitmap cache.

Indeed, this is the case. According to Microsoft's product literature, the TSE client reserves, by default, 1.5MB of memory for a bitmap cache using an LRU eviction policy [13]. It is used to store icons, button images, glyphs, and the occasional animation frame.

To further study bitmap cache handling of animations, we created a set of custom controlled animations. The GIF Construction Set allowed us to specify the number of frames in the animation as well as the delay between frame displays. Our test files were spinning animations of a "Dateline NBC" logo downloaded from the web. They were 119x119 pixels in size with a color depth of 8bpp. The animations in our test set differed from one another only in the total number of frames and in the delay between frames.

Overflowing the Cache

First, we show that as long as cache occupancy is below 100%, RDP activity remains extremely low, but when the cache overflows, RDP activity increases sharply and remains high.

Suppose that the cache is n bytes large, and LRU is the eviction policy. Suppose further that each frame of our animation is, accounting for any compression done by RDP, m bytes. An animation with fewer than n/m frames should therefore fit entirely within the cache and we should see very little network load.

It should also be the case that an animation with $n/m + 1$ frames will generate substantially higher load. If there is one more frame than will fit in the cache, and the eviction policy is LRU, then because of the linear-loop access pattern to the frames, every bitmap request should miss in the cache. At each frame request, the frame needed should have just been evicted to make space for the frame immediately preceding it.

To verify this hypothesis, we created a series of animations whose total number of frames ranged from 25 to 100. Figure 4 supports our hypothesis, showing that while load is just 0.01Mbps for frame counts 10 through 65, it rises sharply to 1.00Mbps for all frame counts above 65.

While LRU may be the appropriate eviction scheme for typical usage, it is exactly the wrong scheme for handling looping animations. Although the cache has room for n/m frames, none of them are in the cache when they are needed. A more intelligent scheme might somehow detect loop patterns and dynamically adjust eviction behavior appropriately.

Bypassing the Cache

Flipbook-style animations using a set of bitmapped frames is not the only way to achieve user-interface animation. Animation may also be generated using graphical primitives like line draws. Animated elements of this type will not be aided by the bitmap cache and can generated considerably more traffic.

As a simple test, we ran the "Beziers" screensaver in Windows which draws a pixel-wide loop composed of 10 bezier curves. The loop is continuously bent and twisted as it is translated across the screen. Although we cannot verify with complete certainty that line draw primitives are being used and that the bitmap cache is of no help, the data seem to support such a conclusion. Activating this screensaver generates 96.42 KBps of network load.

While administrators of TSE will almost certainly want to disable screensaver usage, this experiment shows that any dynamic user interface element relying not on bitmaps but on vector- or primitive-based graphics will likely generate considerable load on the network.

4 Latency Characteristics

In the previous section, we examined the resource utilization characteristics of TSE. To end-users, load is an abstract concept only relevant to them when it begins to affect their interaction with the system. In this section, we identify and quantify situations in which server and network load translate to user-perceived latency. Specifically, we discuss the latency produced when the processor, memory, and network are saturated.

4.1 Processor

First, we evaluate TSE's ability to maintain interactivity under heavy processor load. We generated server-side load using a C program executing a trivial tight loop. Since the code never yields the processor, each concurrent instance increases the scheduler queue length by one. We then opened a single remote session and started Notepad. The tester held down a key, engaging character repeat at a rate of 20cps. Using *tcpdump*, we recorded

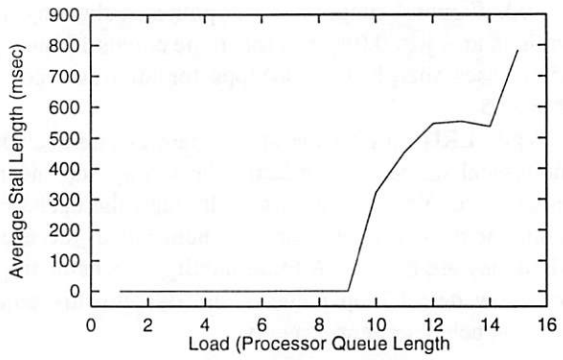


Figure 5: Average stall length experienced by a user given varying CPU loads at the server.

timestamps on the network packets associated with the keystroke and character echo messages.

With a keystroke input rate of 20cps, we would expect the server to respond every 50ms with a screen update message to draw a new character. In our test, we measured the delay between each consecutive screen update message and define the difference between this value and the expected 50ms to be a stall. Figure 5 shows, for varying loads, the average stall length over the course of 60 seconds.

At no load and small loads, TSE performs perfectly, delivering a screen update every 50ms. However, as load approaches 10, the average stall length rises dramatically. At a load of 10, the user spent more than 50% of his time waiting for the server to respond.

In 1993, Evans et. al at SunSoft optimized the System V, R4 scheduler with an eye towards interactivity rather than fairness or real-time requirements [6]. Using a methodology similar to ours, they first demonstrated that in a system based on an unmodified SVR4 kernel, keystroke handling latency increases with load just as TSE does here. They attribute this effect in SVR4 to the lack of protection for interactive processes in the scheduler. Their modified kernel, however, handled load more gracefully, with keystroke handling latency remaining constant and small as load increased. A similar modification to the NT kernel might help it retain good interactive characteristics even when processor utilization is at 100%.

4.2 Memory

High page demand on the server will typically force non-active processes to be paged to disk. This behavior can be particularly damaging to interactivity in the following scenario. An interactive user may load a document into an application but then stop interacting with it for several minutes as he thinks and reads the document on-screen. During this time, high page demand on the server may force his application out to disk. When he reaches again for the keyboard to scroll down, there will be significant lag as his process is paged back into memory.

To measure this effect in TSE, we opened an instance of Notepad remotely. We then started and let run for 30 seconds on the server a process that sequentially touches a heap of bytes whose total size exceeds the available physical memory. After those 30 seconds, we input a single keystroke into Notepad and measured the time it took for the server to respond with a screen update. We report the results for two different load conditions in the table below.

	Warm-start delay (ms)		
Page Demand	Min	Max	Avg
less than 100%	50	50	50
more than 100%	2,400	11,900	4,000

Under heavy page demand, TSE as it runs on our *server* configuration took as long as 12 seconds to reload the process and service the request. With the amount of memory shipping in typical system configurations, today's users of single-user operating systems probably rarely encounter paging behavior, and waiting for a process to reload from disk is likely to be a foreign and highly irritating experience.

Evans et. al also discussed and demonstrated this effect in the SVR4 kernel. Their solution was to throttle non-interactive processes with high page demand, favoring interactive processes instead. Clearly, a memory scheduling scheme better adapted to interactive usage would help the TSE kernel perform better under high memory load.

4.3 Network

With TSE's introduction, Windows programmers must consider the fact that their user interfaces may be exported to remote clients over a network.

The mere introduction of a network link introduces latency. A 10Mbps link can transmit 1.25KB per millisecond. Previous work found that latencies exceeding 100ms begin to be noticeable [15]. Therefore, visually discrete screen updates that cannot be expressed in RDP in 125KB may appear sluggish. Our subjective experience with TSE was that even with no network load, remote interaction was noticeably more sluggish than local interaction at the console. On anything less than full Ethernet, users can expect poor performance. A 56Kbps modem connection, for example, can only transmit 7 bytes per millisecond. The average display message size in our trace from Section 3.3.1 was 702 bytes, and visually discrete screen updates are rarely composed of just one

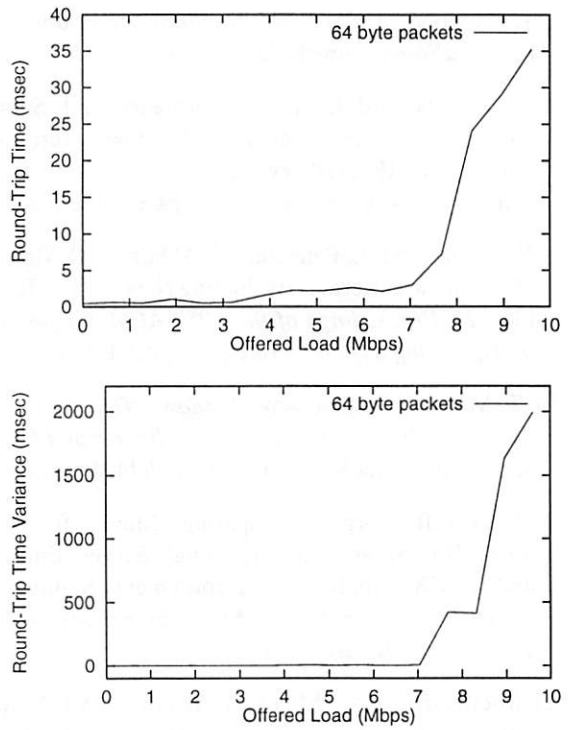


Figure 6: Host-to-host round-trip times (latency) and host-to-host round-trip time variance (jitter) as functions of load.

display message. Therefore, few, if any, operations over such a connection will appear smooth.

Latency also increases dramatically when the network is saturated. To demonstrate, we produced synthetic TCP/IP traffic on our testbed. Figure 6 shows the effect of load on packet latency and jitter (latency variance). For each load level, we ran *ping* for 60 seconds and took the average and variance in round-trip-time (RTT) for all packets sent. We used a packet size of 64 bytes, which is roughly the size of a keystroke control message. So, the latencies we observed give a realistic lower-bound on the additional latency introduced by network saturation.

As the figure shows, while the network is not saturated, RTT remains low and almost perfectly consistent. However, as the network nears saturation, performance suffers dramatically. The 35ms delay induced at 9.6Mbps load is considerable with respect to known levels of human latency tolerance [15]. The inconsistency of the latency, a phenomenon known as jitter, only compounds the negative impact of network saturation.

5 Related Work

We found three technical documents on TSE performance. All are white papers, one published each by Microsoft, Hewlett-Packard, and Compaq [1, 8, 11]. They are all similar, reporting maximum concurrent usage limits for varying combinations of user profiles and server configurations. Their results have limited utility for three reasons. First, they provide little explanation as to the underlying principles of how load is generated, making it difficult to extrapolate limits for configurations other than the few they list. Second, their analyses focus almost exclusively on throughput rather than latency. Third, their user behavior profiles do not include increasingly common tasks like web browsing, which we have shown can produce substantially higher network load.

The Compaq paper stands apart from the HP and Microsoft papers by giving some justification as to why the proffered usage limits are relevant. They identify latency as the key metric for performance:

> The maximum number of clients at an acceptable performance level was determined to exist just prior to a delay in the response time at the client. That is, when the script was executing, there was a pause between entries of characters. Processor queue length was an essential performance factor that was critical in fixing this delay. Processor queue length is the instantaneous length of the processor queue in units of threads. A processor queue length of 12 corresponded to a delay in typing and was consequently used to gauge maximum client support.

They seem to say here that the usage limit is reached when adding one more client pushes typing latency above some threshold.

Harwood and Mathers and Genoway authored books aimed at system administrators on configuring, using, and testing TSE [7, 10]. Danskin and Hanrahan wrote several papers on profiling the X protocol, which provides functionality similar to RDP for Unix systems [3, 4]. Endo et. al and Evans et. al, as discussed earlier, both wrote papers of a rare breed in OS research that address the issue of user-perceived latency [5, 6].

6 Conclusions

We have analyzed TSE's resource consumption and latency characteristics.

The transition from NT to TSE has introduced additional compulsory load. TSE has 71% more CPU activity when idle than NT Workstation 4.0 and each user session consumes a minimum of 2.5MB of memory. Typical applications, even with code page sharing, consume at least 1MB for each instance. While RDP is efficient for simple

operations like typing and mouse movement, more sophisticated user interaction like web browsing and document scrolling can generate surprisingly high network loads. On 10Mbps Ethernet, this behavior can limit concurrent use to as few as 10 sessions.

In terms of latency, the TSE kernel does not protect interactive processes against high CPU and memory load as well as it could. Evans et. al suggest potential remedies. Network latency even with no load degrades the user experience. Given the average RDP message size, TSE remote sessions would likely be difficult to use over modem links with speeds on the order of tens of Kbps. Without improving some of the latency characteristics of TSE, adopters will need to readjust the interactivity expectations they developed while using single-user systems with copious memory and fast, local video.

7 Acknowledgements

We would like to thank our shepherd, Sam Leffler, the anonymous referees of the program committee, Mike Jones, and Chin-hoa.

References

[1] Compaq Corp. "Performance and Sizing of Compaq Servers with Microsoft Windows NT Server 4.0, Terminal Server Edition," `http://www.compaq.com/support/techpubs/whitepapers/ecg0680698.html`

[2] B. Conner, L. Holden. "Providing A Low Latency User Experience In A High Latency Application," *Proceedings of the 1997 ACM Symposium on Interactive 3D Graphics*, pp. 45-48.

[3] J. Danskin, P. Hanrahan. "Profiling the X Protocol," *Proceedings of the 1994 ACM Conference on Measurement and Modeling of Computer Systems*, pp. 272-273.

[4] J. Danskin, P. Hanrahan. "Higher Bandwidth X," *Proceedings of the 1994 ACM Multimedia Conference.*

[5] Y. Endo, Z. Wang, B. Chen, M. Seltzer. "Using Latency to Evaluate Interactive System Performance," *Proceedings of the 1996 USENIX Symposium on Operating System Design and Implementation*, pp. 185-199.

[6] S. Evans, K. Clarke, D. Singleton, B. Smaalders. "Optimizing Unix Resource Scheduling for User Interaction," *Proceedings of the 1993 USENIX Summer Technical Conference.*

[7] T. Harwood. *Windows NT Terminal Server and Citrix MetaFrame*, New Riders, 1999.

[8] Hewlett-Packard Corp. "Performance and Sizing Analysis of the Microsoft Windows Terminal Server on HP NetServers," `http://www.hp.com/netserver/techlib/perfbriefs/`

[9] N. Hutchinson, L. Peterson, M. Abbott, S. O'Malley. "RPC in the x-Kernel: Evaluating New Design Techniques," *Proceedings of the 1989 ACM Symposium on Operating Systems Principles*, pp. 91-101.

[10] T. Mathers, S. Genoway. *Windows NT Thin Client Solutions: Implementing Terminal Server and Citrix MetaFrame*, MacMillan Technical Publishing, 1999.

[11] Microsoft Corp. "Comparing Microsoft Windows NT Server 4.0, Terminal Server Edition, and UNIX Application Deployment Solutions," `http://www.microsoft.com/ntserver/zipdocs/ts_unix.exe`

[12] Microsoft Corp. "Microsoft Windows NT Server 4.0, Terminal Server Edition: Bringing Windows to Desktops That Can't Run Windows Today," `http://www.microsoft.com/ntserver/zipdocs/tsoverview.exe`

[13] Microsoft Corp. "Microsoft Windows NT Server 4.0, Terminal Server Edition: An Architectural Overview," `http://www.microsoft.com/ntserver/zipdocs/tsarchitecture.exe`

[14] Microsoft Corp. "Microsoft Windows NT Server 4.0, Terminal Server Edition — Capacity Planning," `http://www.microsoft.com/ntserver/zipdocs/tscapacity.exe`

[15] B. Shneiderman. *Designing the User Interface*, Addison-Welsey, 1992.

[16] D. Solomon. *Inside Windows NT: Second Edition*, Microsoft Press, 1998.

HACC: An Architecture for Cluster-Based Web Servers

Xiaolan Zhang, Michael Barrientos, J. Bradley Chen, Margo Seltzer
Division of Engineering and Applied Sciences, Harvard University

Abstract

This paper presents the design, implementation, and performance of the Harvard Array of Clustered Computers (HACC), a cluster-based design for scalable, cost-effective web servers. HACC is designed for *locality enhancement*. Requests that arrive at the cluster are distributed among the nodes so as to enhance the locality of reference that occurs on individual nodes in the cluster. By improving locality on individual cluster nodes, we can reduce their working set sizes and achieve superior performance for less cost than conventional approaches. We implemented HACC on Windows NT 4.0 and evaluated its performance for both static documents and workloads of dynamically generated documents adapted from logs of commercial web servers. Our performance results show that HACC's locality enhancement can improve performance by up to 121% for our stochastically generated static file case, by up to 40% for our trace-based static file case, and by up to 52% for our trace-based dynamic document case, compared to an IP-Sprayer approach to building cluster-based web servers.

1. Introduction

To handle the ever-increasing population of World Wide Web users, busy Web sites are frequently hosted on a cluster of computers. The load on these servers is further exacerbated by the trend towards *Web Application Servers (WAS)*, which generate documents on the fly, requiring a great deal of compute power on the servers. Clustered web servers are a natural solution to scaling the server for arbitrarily heavy loads. Researchers have studied clustered web server architectures extensively [DKM96, FGC97, KMR95, PAB98]. However, much of their research effort has been directed at support for static files. Anecdotal evidence suggests that the performance issues for WAS are more complex and intrinsically different from those of static file servers [CDW97]. WAS typically consist of a collection of distributed backend servers remotely connected to a web front-end. The performance of a WAS is usually limited either by the network delay between the front-end and the backend server, or by backend computing power [Ten99]. We call this "backend limited" as opposed to the conventional static file case where the server is usually "memory bound" or "disk bound", limited by the number of open sockets it can support. In this paper we present the Harvard Array of Clustered Computers (HACC), a cluster-based design to enhance performance for both static file service and WAS, with WAS as our target domain.

The conventional cluster-server approach puts a router or "IP-Sprayer" between the Internet and a cluster of web servers. The job of the router is to spread the load evenly over the nodes in the cluster. A number of commercial products [Cis96, Che97] employ this approach to distribute web site requests to a collection of machines, typically in a round-robin fashion while attempting to preserve affinity between users and server nodes. The purpose of the affinity is to give better behavior for web servers that maintain state about connected users. Increasing the aggregate performance of the cluster is simply a matter of adding more nodes, with scalability limited only by the capacity of the router.

This simple approach to clustering does a good job of addressing the scalability problem, but it is not a panacea. For example, a server node for a large or complicated web site might require a large amount of physical memory in order to handle requests efficiently. Each node added to the system will be responsible for the same document store, and so will require the same large physical memory. The result is that either the server nodes are expensive (they need a lot of memory), or they are slow (so you need many of them) or both. Overall this leads to an inefficient use of resources. Figure 1(a) shows a schematic representation of this situation. Notice that each node in the cluster is responsible for the same working set, namely the active elements of the entire document store.

HACC eliminates the inefficiencies in this system by two means: locality enhancement and dynamic load balancing. Rather than distributing requests in a round-robin fashion, HACC distributes requests so as to enhance the inherent locality of the request streams in the server cluster. We refer to this modified sprayer as a *Smart Router*, illustrated in Figure 1(b). Instead of being responsible for the entire working set, each node in the cluster is responsible for only a fraction of the document store. The size of each node's working set decreases each time a node is added to the cluster, resulting in more efficient use of resources at each

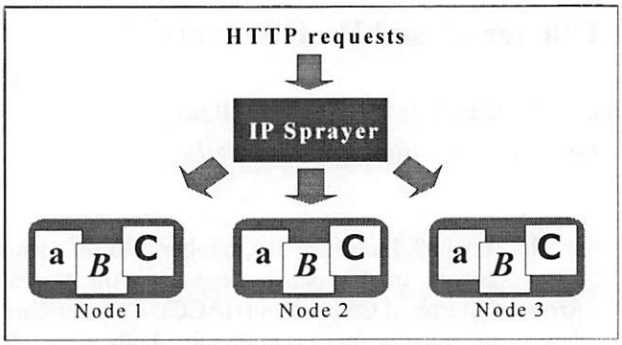


Figure 1(a). IP-Sprayer Architecture

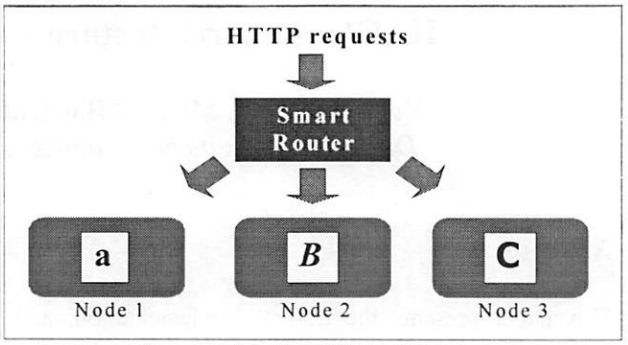


Figure 1(b). HACC Architecture

node. The Smart Router also uses an adaptive scheme to tune the load presented to each node in the cluster based on that node's capacity, so that each node is assigned a fair share of the load. For popular pages, say *Hot Site of the Day*, the Smart Router could direct requests for that page to multiple cluster nodes[1], preventing it from overloading any single node.

The remainder of this paper is organized as follows: Section 2 discusses related work. Section 3 describes the implementation of HACC on Windows NT 4.0. Section 4 presents experimental results. Section 5 discusses future research directions and Section 6 concludes.

2. Related Work

A number of commercial IP-Sprayer products are currently available. One of the better-known products is the Cisco LocalDirector [Cis96]. In addition to performance and scalability, Cisco's product aims to provide failure recovery by detecting long response times that might signal an overloaded or malfunctioning server.

Microsoft's Windows NT Load Balancing Service (WLBS) [WLBS] represents a distributed version of the IP-Sprayer. WLBS functions as a filter between the NIC driver and the TCP/IP protocol stack. It maps incoming requests to the cluster node based on the source IP address and port number and only passes packets that are destined for the local node to the upper network layer. The cluster nodes exchange messages periodically to maintain a coherent view of the mapping and the cluster status. Since the mapping algorithm is fully distributed, this approach removes the single point of failure of the IP-Sprayer. However, as with other IP-Sprayer systems, it does not take advantage of content locality of web requests.

IBM researchers [AS92, DKM96] proposed a variation of the IP-Sprayer approach, called the *TCP router* approach. The TCP router is just like a traditional IP-Sprayer except that the reply sent back by the cluster node bypasses the TCP router and goes directly to the client making the request. This approach requires modifications to the kernel code of each server node in the cluster.

Fox *et al.* [FGC97] provide an interesting overview of the design space for cluster systems, and describe TranSend and HotBot, two cluster-based Internet services implemented at UC Berkeley and Inktomi Corporation. There are a number of important differences between HACC and the Berkeley/Inktomi services. First, HACC is designed to operate with off-the-shelf web server software. TranSend and HotBot are substantially or completely custom web-server software. Second, load balancing in TranSend and HotBot appears to be statically configured, in contrast to the dynamic load balancing in HACC.

The LARD system, developed jointly by Rice and IBM [PAB98], incorporates similar ideas to HACC. In a LARD configuration, the front-end directs incoming requests to back-end cluster nodes based on the content of the requests, and it assumes the role of load balancing. Simulation results and measurements on a prototype implementation show substantial performance enhancement over conventional cluster approaches. However, their work appears to focus entirely on the static file case.

The basic idea of HACC also bears some resemblance to that of affinity-based scheduling schemes for shared-memory multiprocessor systems [TG89, VZ91, SL93], which try to schedule a task on a processor where relevant data already resides.

[1] We assume that the document store is read-only. Mutable document stores introduce a number of interesting issues. Although we believe these issues can be handled in our design, we have focused on the read-only case for this paper.

3. Implementation

The main challenge in realizing the potential of the HACC design is building the Smart Router, and within the Smart Router, designing the adaptive algorithms that direct request streams at the cluster nodes based on the locality properties and capacity of the node. Additionally, the Smart Router must be robust and efficient enough to handle a large number of cluster nodes without becoming the bottleneck in the system. Another challenge is creating a suitable request stream for evaluating HACC performance; therefore, after describing the Smart Router, we also include a brief discussion of DBench, the benchmark used to evaluate our system.

3.1. Smart Router Implementation

The Smart Router implementation is partitioned into two layers, the High Smart Router (HSR) and the Low Smart Router (LSR). As the names suggest, the LSR corresponds to the low-level, nuts-and-bolts kernel-resident part of the system[2], whereas the HSR implements the high-level, user-mode "brains" of the system. This partitioning encourages a separation of mechanism and policy, with mechanism implemented in the LSR and policy in the HSR.

The LSR

The LSR encapsulates the networking functionality required by our design. It is responsible for TCP/IP connection setup and termination, for forwarding requests to cluster nodes, and for forwarding result documents back to clients. Apart from these functional requirements, the main requirement of the LSR is performance — it is on the critical path of every request handled by the HACC cluster and will generally determine the degree of scalability within the cluster.

Our in-kernel LSR is implemented as a Windows NT 4.0 device driver that attaches to the top of the TCP transport driver. The upper edges of all NT transport drivers have an abstract interface known as the Transport Driver Interface (TDI). This layer of abstraction allows us to implement the LSR on top of the TCP/IP transport layer without any modification to the Windows NT networking subsystem. Installing the LSR is just like installing an ordinary device driver. This simplifies the design of the LSR since it does not need to handle any protocol-related issues. Note that this new layered driver is needed only on the Smart Router. Server nodes in the cluster run entirely off-the-shelf software and do not need any new or modified network drivers.

The LSR listens on the well-known web server port for a connection request. When a connection request is received, TCP passes a buffer to the LSR containing the HTTP request. The URL from the request is extracted and copied to the HSR[3]. The LSR enqueues all data from this incoming request and waits for the HSR to indicate which cluster node should handle the request. When the HSR identifies the node, the LSR establishes a connection with it and forwards the queued data (including the URL) over this connection. The LSR continues to ferry data between the client and the cluster node serving the request until either side closes the connection.

This design has some important consequences. As we are maintaining all open TCP connections in the Smart Router, a criticism is that the router immediately becomes the bottleneck, requiring as much networking resources as all the cluster nodes combined. However, we find that in our target WAS domain, managing network state is not the system bottleneck. Each dynamic request requires a significant amount of computation on the server, such that the networking system is not stressed handling incoming packets (cf. sections 4.4 and 4.5).

The HSR

The job of the HSR is to monitor the state of the document store, the nodes in the cluster, and properties of the documents passing through the LSR. It then must use this information to make decisions about how to distribute requests over HACC cluster nodes. To date we have implemented two decision distribution algorithms, one modeling a tree-based name space such as would be appropriate for static file service and one for the document store used by Lotus Domino.

To support the tree-based namespace, the HSR maintains a tree that models the structure of the document store. Leaves in the tree represent documents and nodes represent directories. As the HSR processes requests, it annotates the tree with information about the document store to be applied in load balancing. This

[2] An earlier version of our system used a user-level LSR, similar to proxy server implementations. However, the overhead of repeated crossings of the kernel/user boundary became a significant bottleneck.

[3] The Windows NT 4.0 version of the TCP/IP TDI layer does not support zero copy when data is passed up from the TDI layer to the LSR. We expect this feature to be supported in the future versions of NT, which should produce a significant improvement in LSR performance.

information could include node assignment, document sizes, request latency for a given document, and, in general, sufficient information to make an intelligent decision about which node in the cluster should handle the next document request. In our prototype implementation, load balancing is performed on a per cluster node base. Therefore, only node assignment information is recorded.

When a request for a particular file is received for the first time, the HSR adds nodes representing the file and any newly reached directories to its model of the document store, initializing the file's node with its server assignment. In our current HACC prototype, incoming new documents are assigned to the least loaded server node. After the first request for a document, subsequent requests will go to the same server, improving locality of reference.

However, the tree-structured name space only works for the case when the structure of the document store is hierarchical. Some WAS platforms, such as Lotus Domino, a web-server product from IBM Lotus, embed keys or request parameters into URLs, requiring further semantic analysis of the URL in order to model the structure of the document store. A Domino URL, for example, is composed of three fields: host name, Notes object, and action: *http://host_name/Notes_-object?action*. The host name is the name of the web site. The Notes object field identifies a Notes object within a database, typically a database view followed by a document that belongs to the view. The action field denotes the Lotus command to be activated on the Notes object. Typical actions are "OpenDocument", "OpenNavigator" and "OpenView". Actions can have parameters separated by the "&" character. The HSR extension for Domino incorporates the same tree-structured name space model for Notes objects and enhances it with the "action" information. The model only tracks down the hierarchy to the database view level (as opposed to individual document level). The HSR decides where to forward the request within the cluster based on the Notes object and the requested action.

3.2. Dynamic Load Balancing

Dynamic load balancing is implemented using Windows NT's Performance Data Helper (PDH) interface [PDH98]. The PDH interface allows one to collect a machine's performance statistics remotely, thus relieving us from the burden of implementing a monitoring agent on each cluster node. The only monitoring agent needed is the one on the Smart Router. Another advantage of PDH is that it allows web application developers to add application-specific performance objects and counters that can be retrieved the same way as system performance counters using a set of well-defined APIs.

When the Smart Router starts, it spawns a performance-monitoring thread that collects performance data from each cluster node at a fixed interval. The performance data is used for load balancing in two ways by the HSR. First, a least loaded node is identified and new (unseen) requests are assigned to the least loaded node. Second, when a node becomes overloaded (i.e., its load exceeds that of the least loaded node by a certain amount), the HSR tries to offload a portion of the documents for which the overloaded node is responsible to the least loaded node.

The monitoring thread collects each node's load statistics, such as CPU utilization, disk activity, paging activity, number of outstanding web requests in the queue, etc., and combines these performance metrics into a single load indicator using a weighted average. In our prototype implementation, we use two performance metrics: CPU utilization and bytes transferred to/from disk per second. The load is calculated using the following formula:

$$load = weight_{CPU} \times load_{CPU} + weight_{disk} \times load_{disk}.$$

For the static file workloads, $weight_{CPU}$ is set to zero and $weight_{disk}$ to one, since disk activity is the dominant factor of a server's load. For the Domino workload, both $weight_{CPU}$ and $weight_{disk}$ are set to ½, since a server node should be balanced between the complexity of the tasks it handles and the working set size[4]. In spite of its simplicity, this formula works well for our test cases.

It is an interesting research question to determine how to combine a set of performance statistics into a single metric that reflects the cluster node's real load in the context of the particular web application. This is an area of further research and beyond the scope of this paper.

3.3. DBench

DBench [CDW97] is designed specifically for evaluating WAS performance. The D in DBench is for *dynamic*, emphasizing the key difference between DBench and existing benchmarks — its ability to test WAS performance. DBench is based on a simulator that models the activity of multiple individual users accessing a Web site. DBench supports requests for WAS by replaying the sequence of document requests generated by an actual user and by dynamically

[4] Statistics of disk activity are normalized to the same scale as that of CPU load.

Workload	Description	Server software	Web site size (MB)	Average file size (KB)	Number of files
Static baseline	Simulated clients request static files. Files and scripts are stochastically generated.	Microsoft IIS 4.0	100	280	367
Static FAS	Simulated clients request static files. Based on www.fas.harvard.edu.	Microsoft IIS 4.0	80 (subset)	14	5715
Domino	Simulated clients make requests to Lotus Domino Server. Based on lotus.domino.com.	Lotus Domino 4.6	129 (database)	N/A	N/A
ASP	Simulated clients make ASP as well as static file requests to the Server. Based on www.thecrimson.com.	Microsoft IIS 4.0	54 (static) 7 (database)	12	4495 (including asp files)

Table 1. Workload Descriptions

controlling the number of simulated web users. Internally, users are modeled using a collection of user profiles. For example, the Domino-based DBench test used about 500 profiles, each of which describes the pattern of references that occurred for a real Domino user. The profiles are created by analyzing the access log for a representative web site and extracting the sequence of requests made by each individual that accessed the system. The profiles include the URLs requested by each user as well as timing information that specifies the user pause time between each client request. Each concurrent user in DBench is modeled by replaying the sequence of requests as recorded in a user profile.

DBench reports its results using two primary metrics: *Concurrent Users*, which is to help site managers make capacity planning decisions and *Aggregate Throughput*. DBench also reports statistics such as *Average Request Latency* and *Number of Requests Completed* for each sub-epoch (10 seconds). We use a subset of these statistics to compare performance of HACC with other approaches. DBench measures server performance by gradually increasing the number of concurrent simulated users until one of three conditions occurs:

- the server begins to generate request failures,
- the average time to establish a connection with the server exceeds 3 seconds (1 second for the static file case), or
- the maximum time to establish a connection with the server exceeds 5 seconds (2 seconds for the static file case).

DBench terminates when the number of concurrent simulated users is stable for 100 seconds.

4. Experimental Results

4.1. Methodology

In this section we present experimental results to compare the performance of a HACC cluster to that of a cluster implemented with an IP-Sprayer. For the HACC cluster, the Smart Router distributes requests using the scheme described in Section 3. For the IP-Sprayer, we replace the HSR portion of the Smart Router with a simple round robin request distribution scheme. Readers might notice that this is not a true implementation of an IP-Sprayer, since IP-Sprayers distribute packets at the IP layer. Consequently our IP-Sprayer implementation will have inferior performance compared to commercial implementations. However, as we will describe in section 4.2, for our target domain of WAS, the overheads incurred by the Smart Router module are minimal compared to the request latency and do not affect the significance of the improvements obtained with the Smart Router.

We used four DBench workloads for this evaluation, a stochastically generated static file test, a static file test based on Harvard's *FAS* (Faculty of Arts and Sciences) web site, a WAS test based on Lotus Domino, and another WAS test based on Microsoft ASP [Wei99]. Details are given in Table 1. For the static-FAS workload, due to the excessive number of user profiles, we randomly select a subset of user profiles and use them to drive DBench. Our cluster consists of four Hewlett Packard Netserver E40 uniprocessors, each with a 200MHz PentiumPro processor with a 256KB 2nd level cache. Three of them are used as cluster nodes and one is used as the Smart Router. To test the capability of HACC to work with uneven cluster node capacities, we intentionally configured the cluster nodes with different memory sizes. Two of the cluster nodes and the Smart Router have 64MB of main memory, and the third cluster node has 32MB. All systems run Windows NT 4.0 updated with Service Pack 4. For the cluster interconnect, we use 100Mb/s switched Ethernet, with a Hewlett Packard AdvanceStack Switch 800T. Unless otherwise specified, all results reported in this paper are the average of three runs preceded by a warm-up run.

We are aware that Domino is not the most popular web server product on the Windows NT platform and our workloads are relatively small. Obtaining large WAS workloads is extremely difficult, since performance evaluation for WAS requires the original contents of the web site (as opposed to the static file case where the document store can usually be regenerated from the web log), and the large web sites in which we are interested are unwilling to give us the contents due to privacy issues. However, as the workload does not fit in the capacity of a single cluster node, we believe that the techniques demonstrated in this paper are readily applicable to larger workloads.

Section 4.2 presents measured overhead of the Smart Router and analyzes the potential overhead of an IP-Sprayer implementation. Section 4.3, 4.4 and 4.5 compare the performance of HACC and an IP-Sprayer using the two static files workloads, the Domino workload and the ASP workload, respectively.

4.2. Overhead of the Smart Router

To quantify the overhead of the Smart Router, we measured the latency of static documents for documents ranging from 512 bytes to 2MB in size under two situations: (1) send the request directly to the web server (Direct); (2) send the request via the Smart Router (Via Smart Router). To isolate the cost of the decision process in the Smart Router from the cost of the extra network hop, we also measured the latency of using a Windows-NT based IP router, omitting the decision that must be made by the Smart Router. Figure 2 depicts the three different scenarios[5].

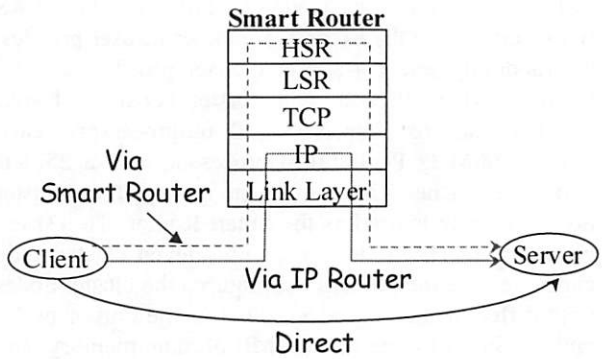


Figure 2. Overhead Measurement under Three Different Situations

[5] To isolate monitoring overhead from that of the request-handling overhead of the Smart Router, overhead was measured with load monitoring disabled. Since load monitoring occurs at infrequent intervals (once per 200 ms), we do not believe that it would affect our results.

The overhead is defined to be the difference between the "Direct" case latency and the latencies of the other two cases. Figure 3 shows the results. The difference of latencies between the "Smart Router" case and the "IP Router" case is the actual cost incurred by the additional TCP connection handling and the decision process in the Smart Router.

To better understand the sources of latency, we split the request latency into two types: fixed overhead and per-byte overhead. Per-byte (or per-packet) overhead includes the cost of going up/down the IP stack. For the Smart Router, there is also an additional per-byte overhead for copying the data between the two TCP/IP connections. This component will be smaller in a zero-copy implementation of the Smart Router. The fixed overhead includes the overhead of the decision process of where to route the packets, plus, for the Smart Router, the time for TCP connection setup and tear-down. We can express the overhead using the following equation:

$$overhead = overhead_{per-byte} \times number\ of\ bytes + overhead_{fixed}$$

or simply $y=ax+b$. Figure 3 shows the measured request handling overhead for our test cases. It also displays the linear equation fit of the data points that gives the fixed and per-byte overhead. The fixed delay induced by the Smart Router is about 1.4 milliseconds. We consider this overhead tolerable in our WAS target domain. More than half of the per-byte overhead of the Smart Router is due to the IP stack. The per-kilobyte overhead of the IP Router is 0.0132 ms/KB, or about 55% of the per-kilobyte overhead of the Smart Router (0.0242 ms/KB).

In summary, for small files, the fixed overhead of the Smart Router dominates. For large files, the Smart Router is about 10-15% slower than an IP Router. It can be argued that a commercial IP Router, such as Cisco's, would probably perform much better than the NT router, however a similar argument can be made about a commercial implementation of the Smart Router. There are many Smart Router optimizations that currently remain unexplored, due to time constraints.

Notice that this test demonstrates the worst case overhead because the time required to service a static file request at the server is minimal (assuming the file is in cache). For CPU-intensive workloads, such as those supported by Domino, serving the request takes more time at the server. At the same time, the result data sent back to the client is relatively small. Thus the overhead for the additional copying is not significant. These two facts together dramatically reduce the actual overhead

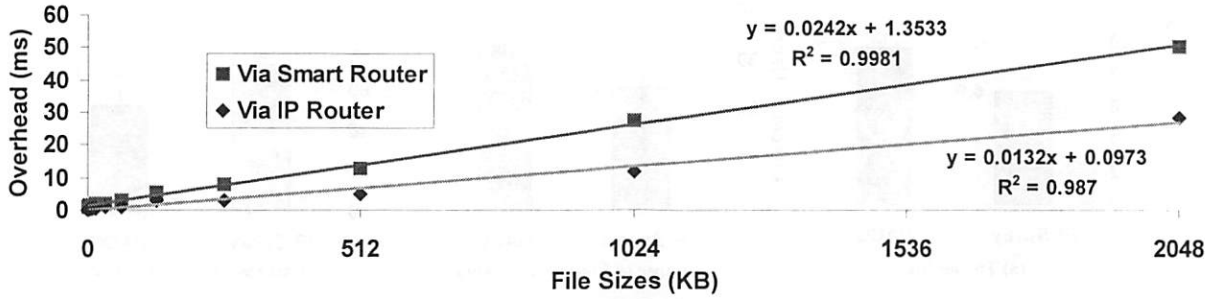


Figure 3. Overhead of the Smart Router for the Static File Case.
As the X-axis is indexed in kilobytes, *a* represents per-kilobyte cost.

as a percentage of the total request service time. For example, in our test benchmark, the average file size for Domino is less than 8KB and the average service time is about 200ms. This results in an overhead of less than 1.6/200=0.8% for the Smart Router. In a real Internet WAN environment, web request latencies are typically dominated by network latencies. This would further reduce the percentage overhead of the Smart Router. Therefore, we conclude that the overhead of the Smart Router is insignificant for CPU intensive workloads or network-latency dominated workloads.

4.3. Static File Results

We use two static file workloads for our HACC evaluation (see Table 1). The first serves as a proof of the basic concept that if the working set of a document store doesn't fit in a single machine's main memory, and the requests can be distributed to preserve locality, then HACC should provide performances superior to that of a conventional IP-Sprayer. For this baseline test case, we use a stochastically generated workload with uniform request distribution to drive DBench. The distribution of the file sizes follows the long tail distribution, i.e., about 40% of the files have sizes between 5KB and 8KB and the rest of the files are scattered between 8KB and 1.4MB. A second static file workload, derived from real logs of the Harvard's FAS web server, is to evaluate how well HACC performs in practice. Figures 4 and 5 give the results for these two test cases.

Figures 4 (a) and (b) show the throughput and the number of concurrent users of the IP-Sprayer and the HACC Cluster for our baseline test. Figure 4 (c) shows the average request latency. The HACC cluster gives consistently better performance than the IP-Sprayer cluster. Each node in the HACC cluster is only responsible for a portion of the file set, as opposed to the entire file set for the IP-Sprayer organization. This leads to a 121% improvement in throughput for HACC. The number of concurrent users also increases by 113%, indicating that the HACC cluster is able to support about 113% more users than the IP-Sprayer architecture. Request latencies also decrease dramatically because of the better locality achieved in the HACC cluster. In the case of HACC, CPU utilization for the Smart Router is about 80%, indicating that the Smart Router is close to saturation as the throughput approaches network limit.

As shown in Figure 5, similar improvements, though smaller in magnitude, are attained for the FAS test case. HACC achieves 40% higher throughput and supports 36% more users. It reduces request latency by 37%. In this case the CPU utilization at the Smart Router is only about 20%. The reason for the less significant improvement compared to the baseline case is that the request distribution for the FAS document store is skewed (as opposed to the uniform distribution in the

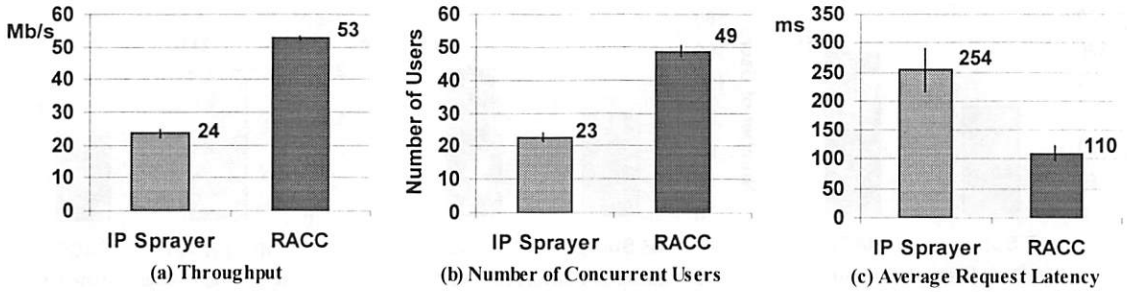


Figure 4. HACC vs. IP-Sprayer Performance for the Baseline Static File Test.
The disk activity for HACC is about 200KB - 300KB per second, and about 700KB - 1MB per second for the IP-Sprayer. The CPU utilization for HACC is about 20% vs. about 10% for IP-Sprayer.

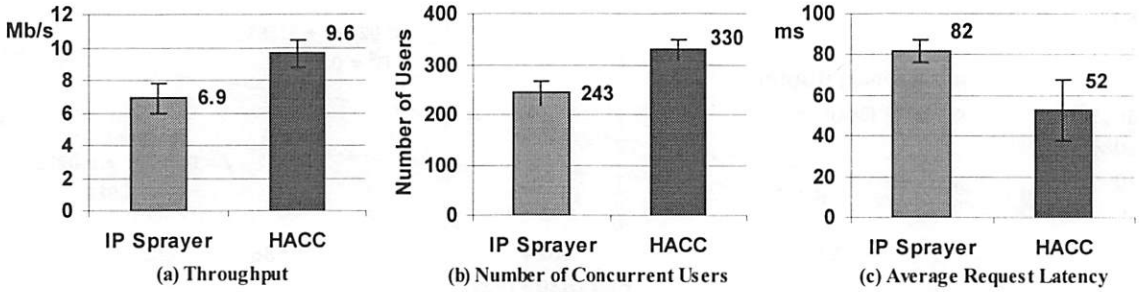


Figure 5. HACC vs. IP-Sprayer Performance for the FAS Static File Test.

The disk activity for HACC is about 200KB - 300KB per second, and about 200KB per second for the IP-Sprayer. The CPU utilization for HACC is about 20% vs. about 10% - 15% for IP-Sprayer.

baseline case). As a result frequently requested files stay in the cache most of the time, even in the IP-Sprayer case, limiting the performance gains attainable by the Smart Router.

Our static file test cases demonstrate that the HACC design can provide substantially improved performance over an IP-Sprayer-based cluster for static file servers.

4.4. Domino Results

The Domino workload is based on the web log and actual document store from domino.lotus.com, a corporate web site for Lotus Notes. While the underlying database is only about 130MB, there is sufficient activity in the web log to keep the server busy. The average request size is about 6KB. Although the log doesn't contain request latency information, our local experiments show that a typical OpenDocument request takes about 200ms and an OpenView request takes between 0.5 to 2.0 seconds when the system is not loaded. The average time gap between subsequent requests is 29.8 seconds, of which a substantial fraction is user "think" time and WAN network latency. For this experiment, the IP-Sprayer is enhanced with a load balancing scheme that always forwards a request to the least loaded node.

Figure 6 shows the performance for the Domino workload. In Figure 6(a) we used *Peak Requests/Minute,* the number of requests completed per minute during the peak period when the number of concurrent users is stable, instead of the *Throughput* metric, since we believe that for the dynamic case, the number of requests a server can handle is a more relevant measure of performance. The HACC cluster delivers over 52% more requests per minute than the IP-Sprayer and supports 46% more concurrent users for this test. Additionally, the average request latency is much smaller for HACC. These results demonstrate that locality enhancement in HACC, even for a mainly CPU-intensive workload, improves performance substantially when the total working set size doesn't fit in a single node's main memory. During all the experiments, the CPU utilization on the Smart Router is only about 1%. Memory usage is also minimal. The cluster nodes show a CPU utilization of 20% - 50% and disk transfers of 400KB - 900KB during the peak period. These data show that for our Domino workload, backend servers are the bottleneck, not the front-end network.

Readers are advised against comparing the absolute numbers of the Domino results with those of the static file cases directly, as these workloads have drastically different characteristics. For example, in the baseline static file case, the scripts used to drive DBench are stochastically generated, and requests from the same

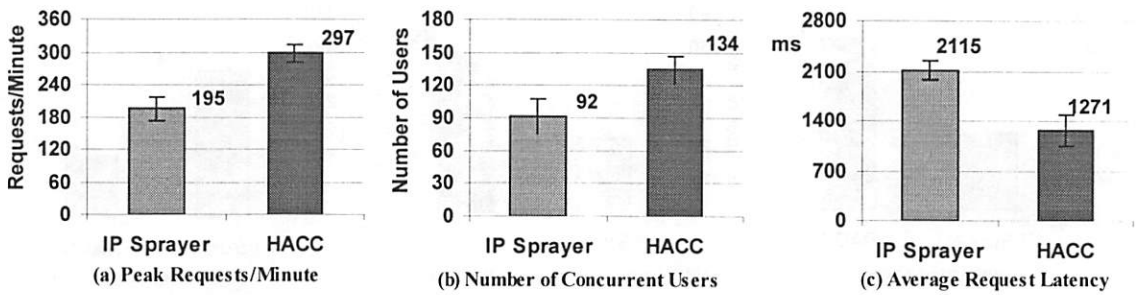


Figure 6. HACC vs. IP-Sprayer Performance for the Domino Test.

The cluster nodes show a CPU utilization of 20% - 50% and disk transfers of 400KB - 900KB during the peak period for both cases.

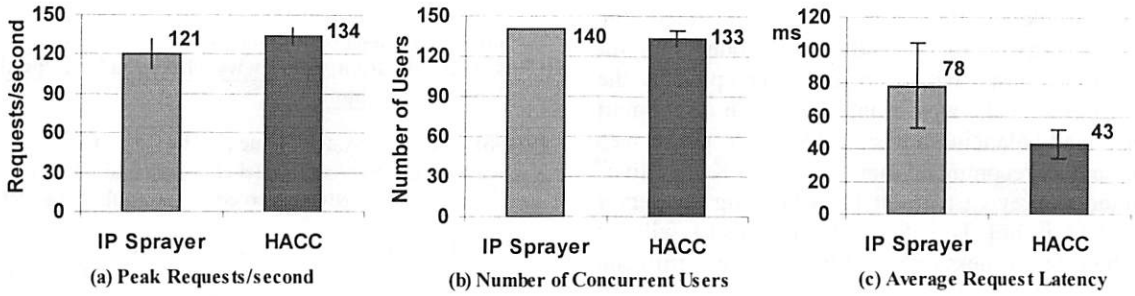


Figure 7. HACC vs. IP-Sprayer Performance for the ASP Test.

For the IP-Sprayer, the CPU utilization of the three cluster nodes is around 50-60%. For the Smart Router, the CPU utilization is about 20% for the node that handles static file requests, and about 40-50% for the other two nodes.

simulated user are 2 seconds apart. In the Domino case, the scripts model actual user behavior and incorporate user "think" time and network latency. The gap between successive requests from the same simulated user is thus significantly larger for the Domino case, resulting in a larger number of concurrent users, even though it takes a much longer time to process a Domino request than a static file request.

4.5. ASP Results

The ASP workload is based on the web log and actual document store from www.thecrimson.com, the online version of *The Crimson* newspaper (see Table 1). Analysis of the web logs reveals that most requests are for static files, and only about 5% of requests are ASP requests. Although only a tiny fraction of the logs are ASP requests, our measurements show that even this low volume of ASP traffic overloads the CPU, long before the network becomes a bottleneck. This confirms our assertion that it is common for WAS environments to be "backend limited."

Since the active working set fits in a cluster node with 64MB of memory, the Smart Router's locality-based request distribution scheme will not offer much advantage over an IP-Sprayer approach. However, we demonstrate here another way of utilizing the Smart Router's content-based routing. For a heavily loaded web server, a CPU-intensive dynamic web request (such as an ASP request) will delay many static file requests, resulting in longer response times for static files. Thus, if we separate ASP requests from static file requests and send them to different cluster servers, we should be able to reduce response time significantly.

We implemented this simple content-based routing scheme and evaluated its performance against an IP-Sprayer approach. The Smart Router forwards static file requests to one of the cluster nodes with 64MB of memory and ASP requests to the other cluster nodes. The results are shown in Figure 7. As expected, the Smart Router approach reduces the average request time by 45%. The IP-Sprayer is able to support slightly more concurrent users because of the perfect load balancing between the three cluster nodes.

This simple experiment demonstrates the flexibility of the Smart Router's content-based routing scheme. Even for a small web site whose working set fits in main memory, the Smart Router can help improve performance.

5. Discussion

There are a number of possible directions for extending and completing the functionality of the system. Mutable document stores present an interesting challenge for HACC. One solution is to offload the consistency responsibilities to backend database servers by using a "three-tier" architecture, with the web servers as a front end for a standard relational database. Use of a HACC cluster as the middle tier of such a three-tier system makes scalable computing available for content access.

A more fundamental issue is scalability of the Smart Router. We conducted a simple scalability test which suggested that our prototype Smart Router can handle between 400 to 500 requests of size 8KB per second. Though not an impressive number for static file web servers, for the Domino Lotus case, this means that it can support around 100 cluster nodes. Therefore, we believe that for web sites that contain a non-trivial portion of dynamically generated documents, the Smart Router will be able to scale.

Furthermore, there are many ways to improve our un-optimized prototype Smart Router implementation. One approach is to implement a TCP connection handoff protocol [PAB98] such that after the Smart Router determines to which node to distribute the request, the TCP connection is handed off to that particular node, which then sends the reply directly to the client, bypassing the Smart Router.

The "Keep Alive" feature of HTTP poses some potential problems. If "Keep Alive" is enabled, the browser is allowed to reuse the TCP connection for subsequent requests, which are not intercepted by the Smart Router. This would interfere with the Smart Router's load balancing decision. However, major web server vendors recommend that the use of "Keep Alive" be limited to prevent a client from hogging the server resources [Apache]. Therefore we expect that it will not affect the effectiveness of Smart Router in a significant way.

6. Summary

In this paper we have presented the HACC architecture and experiments that explore how locality enhancement in HACC improves web-server performance. Experiments with an actual implementation of the HACC cluster on Windows NT show that HACC is able to support more than twice the number of concurrent users in the baseline static file test, 36% more in the FAS static file test, and 46% more in the Domino test than alternative schemes for creating cluster-based web servers. We conclude that the HACC design can be effective for both the static file case and the dynamic case, but in practice, expect it to be most beneficial in the dynamic case where the additional overhead of the Smart Router is tolerable.

7. Acknowledgements

We thank our shepherd Susan Owicki and our anonymous reviewers for their valuable comments. We are grateful to Mark Day at Lotus who answered many of our questions about Lotus Domino. We especially thank Skylar Byrd and Dawn Lee from Harvard Crimson for their generous help in providing the logs and content databases. This research is sponsored in part by Microsoft.

References

[Apache] "Apache Keep-Alive support." This document can be obtained from http://www.apache.org/docs-1.2/keepalive.html.

[AS92] C. Attanasio, and S. Smith, "A Virtual Multiprocessor Implemented by an Encapsulated Cluster of Loosely Coupled Computers." *IBM Research Report RC18442*, 1992.

[CDW97] J. B. Chen, A. Wharton, and M. Day, "Benchmarking the Next Generation of Internet Servers." This document can be obtained from http://www.notes.net/today.nsf by a full text search on "DBench" on the archives of Iris Today.

[Che97] Check Point Software Technologies Inc., "ConnectControl: Advanced Server Load Balancing." Software product. Additional information on this software is available from, http://www.checkpoint.com/products/-floodgate-1/cc.html.

[Cis96] Cisco Systems Inc., "How to Cost-Effectively Scale Web Servers." Packet Magazine, Third Quarter 1996. See http://www.cisco.com/warp/public/784/5.html.

[DKM96] D. Dias, W. Kish, R. Mukherjee, and R. Tewari, "A Scalable and Highly Available Server." In *COMPCON 1996*, IEEE-CS Press, Santa Clara, CA, February 1996, pp. 85-92.

[FGC97] A. Fox, S. Gribble, Y. Chawathe, E. Brewer, and P. Gauthier, "Cluster-Based Scalable Network Services." In *Proceedings of the 16th Symposium on Operating Systems Principles*, ACM, Saint-Malo, France, October 1997, pp. 78-91.

[KMR95] T. Kwan, R. McGrath, and D. Reed, "NCSA's World Wide Web Server: Design and Performance." In *IEEE Computer*, 28(11):68-74, November 1995.

[PAB98] V. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum, "Locality-Aware Request Distribution in Cluster-Based Network Servers." In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, San Jose, CA, October 1998.

[PDH98] "Performance Data Helper", Microsoft Developer Network Platform SDK, Microsoft, July 1998.

[SL93] M. Squillante and E. Lazowska, "Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling." In *IEEE Transactions on Parallel and Distributed Systems,* 4(2):131-143, February 1993.

[Ten99] Jessie Tenenbaum, Microsoft, personal communication.

[TG89] A. Tucker and A. Gupta, "Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors." In *Proceedings of the 12th Symposium on Operating Systems Principles*, ACM, Litchfield Park, AR, December 1989, pp. 159-166.

[VZ91] R. Vaswani and J. Zahorjan, "The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared Memory Multiprocessors." In *Proceedings of the 13th Symposium on Operating Systems Principles*, ACM, Pacific Grove, CA, October 1991, pp. 26-40.

[WLBS] Microsoft Windows NT Load Balancing Service. This document can be obtained from http://www.microsoft.com/ntserver/NTServerEnterprise/exec/feature/WLBS/.

[Wei99] A. K. Weissinger, *ASP In a Nutshell*. O'Reilly, Sebastopol, CA, 1999.

Reducing Startup Latency in Web and Desktop Applications

Dennis Lee, Jean-Loup Baer, Brian Bershad, and Tom Anderson
Department of Computer Science and Engineering
University of Washington, Box 352350
Seattle, WA 98195-2350
{dlee,baer,bershad,tom}@cs.washington.edu

Abstract

Application startup latency has become a performance problem for both desktop applications and web applications. In this paper, we show that much of the latency experienced during application startup can be avoided by more efficiently packing application code pages. To take advantage of more efficient packing, we describe the implementation of demand paging for web applications. Finally, we show that combining demand paging with code reordering can improve application startup latency by more than 58%.

1 Introduction

Program startup latency is an important performance problem for both desktop and web applications. Desktop applications are becoming larger and improvements in disk speeds have not kept up with the improvements in CPU speed. For example, Microsoft Word 2.0 on an Intel 66 Mhz 486 takes about 14 seconds to start, while Microsoft Word 7.0 on a 200 Mhz Pentium Pro occasionally takes almost 17 seconds to start.

The situation is worse for web applications on the Internet. For many users, web applications take a long time (a few minutes) to start because they connect to the Internet through high latency, low bandwidth links (i.e., modems). For example, Vivo, a popular video display and control application, takes over 8 minutes to download over a 56 Kbps modem link and over 2 minutes on a 256 Kbps DSL link. A similar problem occurs in the context of corporate intranets: mobile professionals frequently download new versions of internal corporate web applications and experience frustrating wait times for these applications.

Researchers have recently proposed several ways for improving startup latency including compression [Enst et al. 97, Franz & Kistler 97], non-strict execution [Krintz et al. 98], just-in-time code layout [Chen & Leupen 97], and optimizing disk layout [Melanson 98] .

Our approach is orthogonal and uses *code reordering* [Pettis & Hansen 90] and *demand paging* [Levy & Redell 82] to improve the startup latency of web and desktop applications, and reduce the load on web servers and the network. Our approach essentially improves startup time by improving the effectiveness of demand paging systems: we place procedures that will probably be used by the application into a single contiguous block in the binary. This approach can improve the startup latency of web applications by more than 58% and that of desktop applications by more than 45%.

The rest of this paper

Section 2 presents the motivation and architecture for improving application startup latency. Section 3 describes our experimental methodology and setup. In Section 4, we present the results of our measurements. Finally, Section 5 concludes.

2 Approach

As motivation, we first present the results of profiling several web and desktop applications. These profiles show that existing applications can be better laid out to optimize startup latency. We then

present our architecture for code reordering and our architecture for allowing partial downloads of web applications using demand paging.

2.1 Motivation

Web Applications

Application	Description	Size
envoy	Document viewing control	1.09
scout	VRML parser and renderer	0.98
vivo	Applet for watching movies	0.44
whip	AutoCAD drawing display control	0.49

Desktop Applications

Application	Description	Size
acrobat	Adobe Acrobat Reader 3.0	2.26
netscape	Netscape Navigator 3.1	3.17
photoshop	Adobe Photoshop 4.0	3.65
powerpoint	Microsoft PowerPoint 7.0b	4.36
word	Microsoft Word 7.0	3.78

Table 1: *Web and Desktop Applications. Our web applications consists of four ActiveX [Chappell 96] controls which display various document types. The size column gives the size of the main application binary in megabytes. For web applications, the given size is the uncompressed size of the main application binary and does not include dynamically loaded libraries (DLLs) used by the applications.*

We profiled four web applications and five desktop applications [1] (c.f., Table 1) to determine if there was an opportunity to improve startup time by improving the layout of procedures in a program binary.

Table 2 shows the statistics derived from profiling four web applications. We ran these applications to completion with a typical workload to determine how many procedures in the application are actually used. The table shows that the applications utilize only between 38% (*envoy*) and 84% (*scout*) of the bytes in their program binaries. Typically, web applications download their entire program binaries before starting. The utilization statistics suggest that startup times could be significantly improved if we download only the procedures actually used by the application.

[1] Our tools work only on applications that contain relocation information. We used the latest versions of the desktop applications that contain relocation information. Unfortunately, many new applications ship with relocations stripped.

App.	Kilobytes in Each Component					
	Data		Used Code		Unused Code	
envoy	248	(23%)	164	(15%)	685	(62%)
scout	285	(29%)	262	(27%)	440	(45%)
vivo	274	(62%)	102	(23%)	69	(16%)
whip	197	(40%)	71	(14%)	224	(46%)

Table 2: *Utilization of Web Applications. The percentages in parenthesis show the fraction of the entire binary covered by the particular component. The data column shows the size of all the non-code sections of the binary including section headers. The unused column shows the potential benefit of not downloading the entire application binary.*

Table 3 shows statistics for code pages of different desktop applications during startup. The binary for desktop applications is demand paged so we examine the utilization of the code pages brought in during startup. The table shows that only 26% (*netscape*) to 47% (*word*) of these pages are utilized. Interestingly, *netscape* and *photoshop* touch almost every code page in their main binary during startup suggesting that even with demand paging the entire application is loaded to memory from disk. The low page utilization numbers for all applications suggest that like web applications, startup latency can be improved for desktop applications by bringing in only the actual procedures used by the application.

Application	Code Pages			
	Total	Touched		Utilization
acrobat	404	246	(60%)	28%
netscape	388	388	(100%)	26%
photoshop	594	479	(80%)	28%
powerpoint	766	164	(21%)	32%
word	743	300	(40%)	47%

Table 3: *Desktop Application Profile Results. Touched gives the number of code pages touched during program startup. Utilization gives the average fraction of used procedures in touched code pages. Page fault rates during startup could be reduced by better packing code pages.*

2.2 Architecture

The previous subsection showed that much of the code transferred over the network or transferred from the disk is not used by the application. Our approach aims to transfer only the used procedures in the application.

Figure 1 shows a diagram of the object rewriting phases of our approach. We use profile information to predict with high accuracy which part of the application would be used. Using an object rewriting engine, we then move the likely-used procedures together at the top of the code section, essentially packing pages better to make the demand paging system more efficient. For our experiments, we simply arrange the code section in first-touch order. Ordering using procedure affinity [Pettis & Hansen 90] might be better for locality but first touch order works well enough for our goal of improving startup latency.

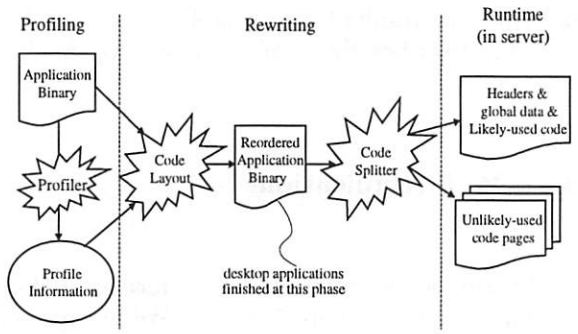


Figure 1: *Object Rewriting Phases*

For desktop applications, the system is done after generating the binary from the code reordering phase. The built-in O/S demand paging system will load in only the fraction of the pages containing the used procedures.

For web applications, the system needs to be able to download only the part of the application required for execution. The code-splitting module splits the binary into (1) a large main binary that contains the data portion of the binary and the likely-used procedures, and (2) several page-sized files containing the unlikely-used procedures. At runtime, when the client requests the web application, the server only transfers the main binary. When the client needs an unlikely-used procedure, it has to request the page-sized file that contains the procedure.

Web Client Architecture

Our client architecture extends demand paging to web applications. This provides a convenient mechanism to detect missing pages and to allow the system to function correctly when control passes to functions that are not present in the initial download.

Figure 2 shows a diagram of what happens on the web client during program runtime. When a web application is accessed by the client browser, only pages containing the data and likely-used portion of the binary are downloaded. The part of the binary that has not been downloaded is marked PAGE_NO_ACCESS by the system.

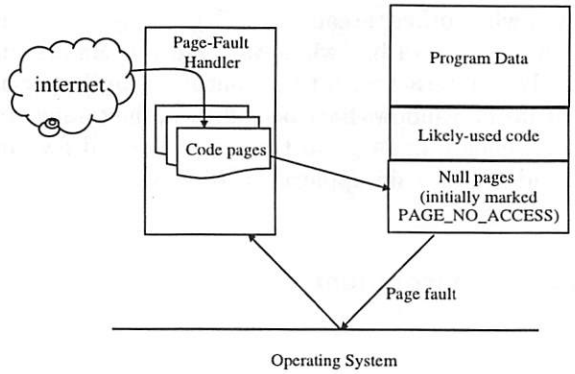


Figure 2: *Web Client Architecture*

If the application transfers control to a page that is marked PAGE_NO_ACCESS, the page-fault handler is invoked. We modified the handler to contact the web server, download the file containing the page, and place the page in the appropriate location in application memory. [2]

3 Experimental Methodology

3.1 Startup Latency

To determine the startup latency, our timing system (1) invokes the application, and (2) simulates a user initiated event by sending a message to an application window. We define startup latency as the time from the invocation of the application to the time the application replies to the message sent by the timing system.

Our timing method works because of the Windows NT event queue model and the way most Windows

[2] As an alternative to downloading individual files each containing a single code page, the client could use the range option in `HTTP 1.1` [Nielsen et al. 97]. This would avoid splitting the application binary into multiple files.

applications are written. Under Windows NT, windows are assigned to threads, and messages are sent to thread-local event queues. Messages are delivered to this queue and do not interrupt the execution of the owning thread. For most applications, threads process their message queue only when they have finished initializing and are ready to respond to user input.

Occasionally, a thread responsible for the main application window will respond to a user message even when other threads are still drawing other windows (e.g., tool bar windows). Since users are unlikely to interact with the application until after all the initial windows have been drawn, the timing system sends a message to the last window drawn instead of the main application window.

3.2 Environment

Our client system was a Pentium Pro 200 system running Windows NT 4.0 Service Pack 3, with 128 MB of memory, and a Seagate ST34371W disk. We used a slightly modified version of Internet Explorer 4.0 as our browser. Our measurements were taken using the processor cycle counter and the performance counters built into Windows NT. All our network measurements were taken on isolated networks with no external traffic.

For our web server, we use Apache 1.3b5 running on FreeBSD 2.2.6. To control the bandwidth and latency between the web server and the client, we installed the dummynet [Rizzo 99] patch to the BSD kernel. Our Internet application experiments looked at a range of bandwidths (from 56 Kbits/second to 3 Mbits/second) and latencies (from 10 ms. to 200 ms.) which cover the range of network conditions on the Internet.

The application server used in our experiments with application startup latency was a Pentium Pro 200 system running Windows NT 4.0 service pack 3 with 64 MB of memory. Unfortunately, we could not control the bandwidth or latency to the application server on NT so our measurements for desktop applications only involve communication on a single shared 10 Mbit Ethernet link.

We used Etch [Romer et al. 97, Lee et al. 98], a binary instrumentation and rewriting engine, to profile and rewrite the applications used in this study. For all our experiments, we profile and reorder only the main application binary. For our prototype implementation, we simulate having an augmented page fault handler using the Windows NT debugger API [Microsoft 98]. The web browser is run in the context of a custom debugger.

4 Results

In this section, we present the results of our experiments optimizing the startup latency. Section 4.1 describes the results for web applications, and Section 4.2 describes the results for desktop applications.

4.1 Web Applications

In this subsection, we show the performance of our optimization on web applications. We first present a performance model describing the startup latency of web applications and show how our optimization improves startup latency. We then compare four different schemes for starting web applications.

4.1.1 Performance Model

Equation 1 is a simple model for predicting the startup latency of web applications:

$$Startup = \frac{Bytes}{Bandwidth} + Requests \times Latency + Overhead \quad (1)$$

where:

- *Bandwidth* and *Latency* are the observed network bandwidth and latency between the client and server,
- *Bytes* is the number of bytes transfered,
- *Requests* is the number of requests for files to the web server, and
- *Overhead* is the fixed overhead of executing instructions to start the application.

Equation 1 suggests that we can improve startup time by reducing the number of bytes transferred and transfer all the needed bytes in a single requests. Unfortunately, we cannot predict with perfect accuracy the actual bytes that the application would use. Different approaches thus have to strike a balance between including as little as possible into the initially downloaded package and paying the cost of extra requests.

Obviously, an approach would be faster than another if it made fewer requests and transferred fewer bytes. However, if an "improved" approach reduces the number of bytes transferred at the cost of more requests then it would be faster than the original approach, if and only if:

$$Startup_{Orig} > Startup_{Imp} \quad (2)$$

$$\frac{Bytes_{Orig} - Bytes_{Imp}}{Requests_{Imp} - Requests_{Orig}} > Bandwidth \times Latency \quad (3)$$

Equation 3 implies that the performance of one approach relative to another is closely related to the bandwidth-latency product. An improved approach may be faster when the bandwidth-latency product is small but the same approach might actually be slower when the bandwidth-latency product is large.

4.1.2 Different Approaches

We compare the performance of four approaches to starting web applications. These approaches examine the performance of demand paging and reordering, and probe the space of trade-offs between (1) eager approaches which download more bytes in a few requests, and (2) lazy approaches which download less bytes in many requests.

- *Original* downloads the entire binary at once. This approach assures that there will only be a single request.
- *Paged* downloads a code page of the binary only when it is needed. All of the program data is still downloaded initially. This approach reduces the number of bytes transferred (i.e., unused pages are not transferred) but may have to pay a high cost because of request latency.
- *Reordered-Paged* is like *paged* in that it downloads a code page only when needed. But this approach first reorders the procedures in the application to more densely pack likely-used procedures into fewer code pages. Compared to *paged*, *reordered-paged* minimizes the number of pages needed by the application, effectively reducing the number of bytes transferred and the number of requests to the server that have to be made.
- *Reordered* initially downloads the likely-used portion of the code section with all the data in the binary. It still has to pay the cost of a request when control transfers to a page that is not in the initially downloaded portion of the code but this would be much rarer. *Reordered* may transfer more bytes than the *reordered-paged* approach since some of the pages in the likely-used portion of the binary may not be used.

We implemented prototypes of each of these approaches. Each prototype reorders, pages, and downloads only the main application binary and not the libraries that the binaries depend upon.

Figure 3 shows the results of our experiment starting applications using the different approaches and network conditions. [3] The figure shows the improvement in startup latency for a workload that is different from the profiled workload used to reorder the binary (c.f., Section 2.2).

We highlight a few trends from the figure:

- *Reordered* almost always does better than *original.*
- For low bandwidth (e.g., 60+Kbps) and low latency (e.g., 10 ms) connections, *paged* does better than *original.* However, as the bandwidth-latency product increases (graphs towards the bottom right), *paged* makes too many requests from the server and doesn't reduce the number of bytes sufficiently to compensate.
- Comparing *paged* with *reordered-paged* shows that demand paging still leaves much room for improvement. In all cases, *reordered-paged* does better than *paged*, especially in the cases of *envoy* and *scout.*

[3] Figure 6 in the Appendix shows the raw startup latency numbers for all network conditions.

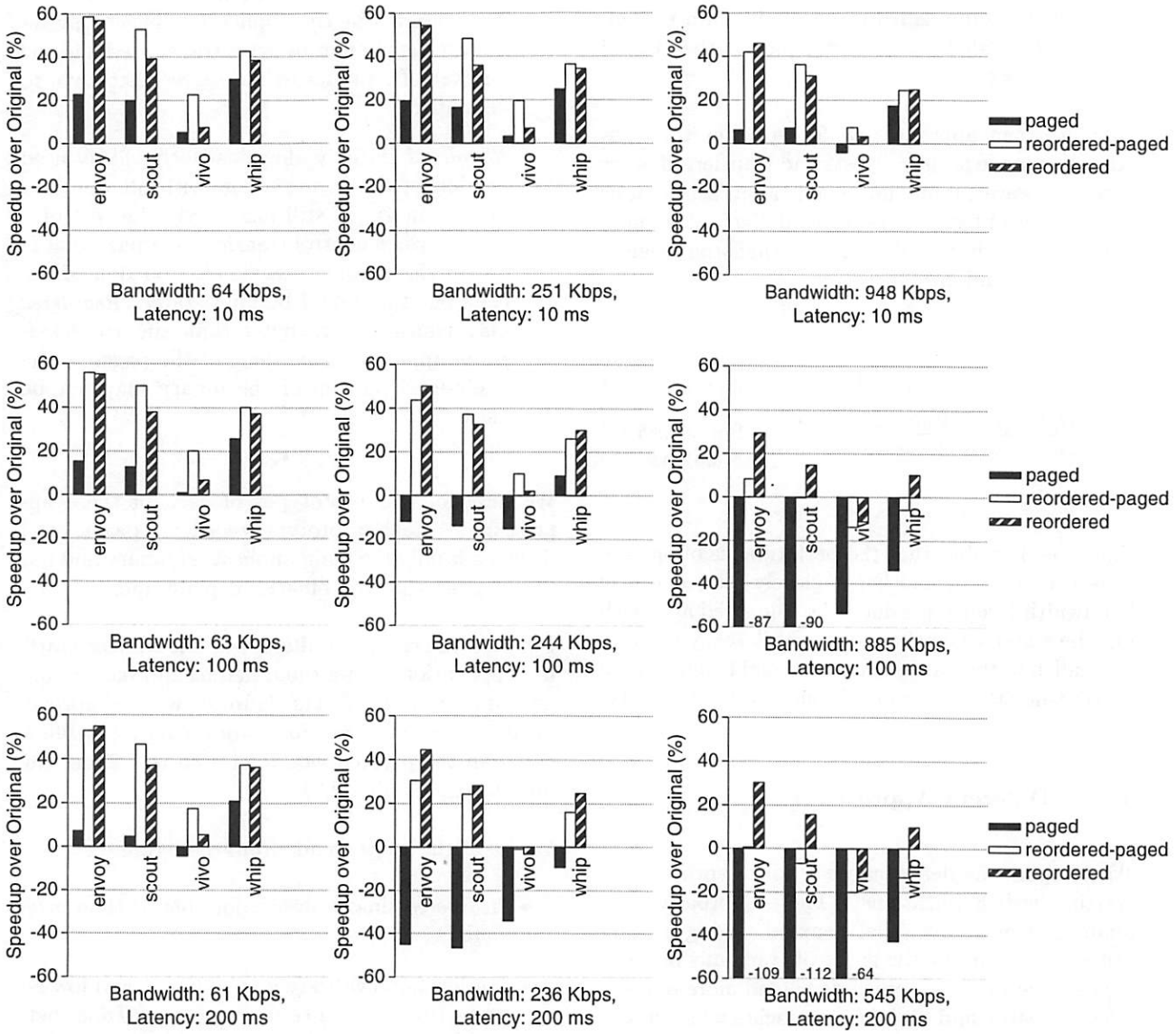


Figure 3: *Web Application Results. Graphs show the improvement in startup latency of web applications under various network conditions. The measured workload was different from the profiled workload used to reorder the web binary. The titles show the measured bandwidths between the client and the server. This is different from the "available" bandwidth (56 Kbps, 256 Kbps, and 1 Mbps) because of the effects of TCP buffering and congestion control [Peterson & Davie 96], and the limitations of dummynet. We attempted to get data points at higher bandwidth-latency products but were limited by the TCP receive buffer size used in Internet Explorer.*

- *Reordered* attempts to do better than *reordered-paged* by reducing the number of server requests. The results show that *reordered* is better than *reordered-paged* for high latency-bandwidth product networks. However, for low-bandwidth networks, the reduced number of bytes transferred by *reordered-paged* make that option better for startup.

The figure also shows a case where our methods are not very effective. For most cases with *vivo*, *reordered* and *reordered-paged* do not do significantly better than *original*. Since *vivo* is already fairly well compacted (84% of the binary is used, c.f., Table 2), we are hard pressed to make the binary more efficient.

Analysis of Downloaded Bytes

Figure 4 shows the breakdown of the bytes downloaded during the startup of the different applications. *Paged* downloads fewer bytes than *original* but generates a fair number of server requests. As shown in Figure 3, this does not work well in the presence of high latency. As expected, *reordered-paged* downloads the least number of bytes and has significantly fewer server requests compared to *paged*. This explains why *reordered-paged* is always better than *paged*.

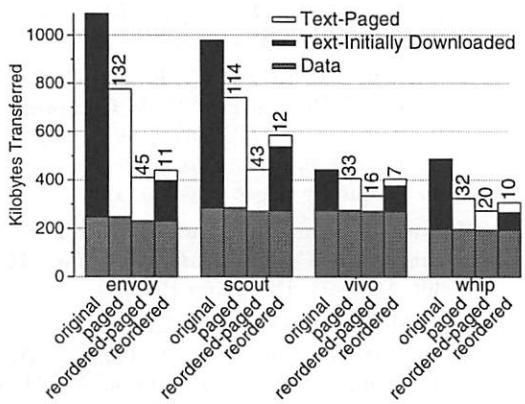


Figure 4: *Download Statistics. Numbers on top of the bars give the number of faults requiring access to the web server. Reordered-paged downloads less bytes than paged and accesses the web server less. Reordered downloads more bytes than reordered-paged but accesses the web server more.*

Reordered reduces the number of server requests further but transfers more bytes. The figure shows that *reordered* still experiences some faults and downloads more bytes than *reordered-paged*. The reason is three-fold. First, the measured workload is different from the profiled workload. We expect that some of the code used in one run would not be used in the other. Second, we profiled the *entire* program run, rather than just startup, to avoid faults even in the middle of the program. Hence, we initially download code that may not be used immediately. This is the case for *reordered* in *scout*. Finally, our profile does not identify the use of data embedded in the code section. This may cause faults on access to code pages containing data.

Reducing the number of bytes transferred has benefits other than reducing the startup latency of web applications. [Banga & Druschel 99, Bradford & Crovella 98] showed that having a large number of open connections on a web server can cause serious performance degradation. By reducing the amount of work that the server has to perform, our techniques can reduce the duration of each connection hence reducing the load on servers serving up these Web applications.

4.2 Desktop Applications

In this subsection, we examine the effect of reordering on the startup latency of desktop applications. We consider the effects of reordering on four different states of the O/S file cache:

- *Boot* - We start the application right after the system boots and the user logs in. The shared libraries required to start the system would be in the file cache but the application binary and other libraries would need to be read in from disk.
- *Warm-local* - the file cache contains the pages from shared libraries but not those of the main application binary. This corresponds to the case when a user starts an application either for the first time or after it has been purged from the file cache. We expect that the shared libraries would be in the file cache due to other applications loading them.
- *Warm-remote* - like *Warm-local* except that the main application binary lives on a remote server. We assume that the application binary is in the file cache of the remote server. This

corresponds to the case when a user starts up an application on a shared file server. Since Windows NT purges its local copy of the file when an application exits [4], we expect this case to happen often in environments with shared application servers.

- *Hot* - the file cache contains all the application and library pages.

We ran the original and reordered application under these four scenarios. To filter out spurious results, we performed each of our measurements at least ten times. We dropped the runs with the highest and lowest times, and report the average of the other runs. Figure 5 shows the results of these experiments.

For all applications and configurations, reordering improves application startup latency. The two *warm* scenarios show that reordering the binary can significantly reduce startup time. It is especially effective for *photoshop* and *word* as they improve their application startup time by almost 2 seconds each. Page fault rates decrease a corresponding amount for the cases where we improve the application startup latency. This verifies that a dominant cost during startup are page-faults.

Since we only reordered the main application binary, we expect to mildly improve the *boot* case. The experiment shows that this is not the case. Many of the shared libraries are loaded into the file buffer cache during boot time as shown by the small difference between the *boot* and the *warm* cases. Hence, reordering just the application binary time shows noticeable improvement even in the *boot* case.

We also included the *hot* case to see if reordering procedures in first-touch order would slow down this case. The figure shows that for the *hot* case, reordering only mildly affects the application startup time, all the differences were less than a tenth of a second.

5 Conclusion

We have implemented a system that uses code reordering and demand paging to improve the startup latency of web and desktop applications. Our measurements on a prototype implementation show that the combination of these optimizations can improve program startup latency of web applications by as much as 58% and desktop applications by as much as 45%.

For web applications, the approaches to improving startup time have to carefully balance the competing requirements of downloading as few bytes as possible and accessing the server the least number of times. This balance is especially important for networks with a high latency-bandwidth product.

[4] Actually, Windows NT purges the local copy of a file when there is no longer an open handle to the file in the local machine [Leach & Naik 97].

References

[Banga & Druschel 99] Banga, G. and Druschel, P. Measuring the Capacity of a Web Server Under Realistic Loads. *World Wide Web Journal*, May 1999. to appear.

[Bradford & Crovella 98] Bradford, P. and Crovella, M. Generating Representative Web Workloads for Network and Sever Performance Evaluation. In *Proceedings of the 1998 ACM SIGMETRICS Internation Conference on Measurement and Modeling of Computer Systems*, pages 151–160, July 1998.

[Chappell 96] Chappell, D. *Understanding ActiveX and OLE.* Microsoft Press, 1996.

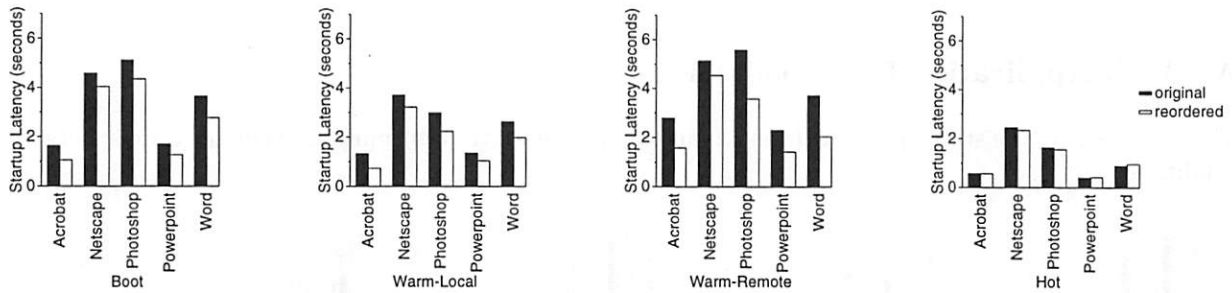
[Chen & Leupen 97] Chen, J. and Leupen, B. Improving Instruction Locality with Just-in-Time Code Layout. In *Proc. of the USENIX Windows NT Workshop*, pages 25–32, 1997.

[Enst et al. 97] Enst, J., Evans, W., Fraser, C., Lucco, S., and T.Proebsting. Code Compression. In *Proc. ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, pages 358–365, 1997.

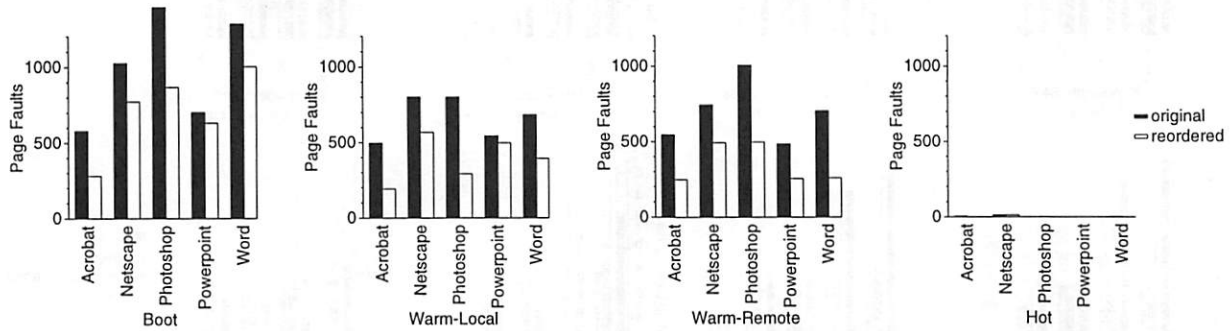
[Franz & Kistler 97] Franz, M. and Kistler, T. Slim Binaries. *Communications of the ACM*, 40(12):87–94, December 1997.

[Krintz et al. 98] Krintz, C., Calder, B., Lee, H., and Zorn, B. Overlapping Execution with Transfer Using Non-strict Execution for Mobile Programs. In *Proc. 8th Int. Conf on Architectural Support for Programming Languages and Operating Systems*, pages 159–169, 1998.

[Leach & Naik 97] Leach, P. and Naik, D. A Common Internet File System (CIFS/1.0) Protocol, December 1997. Internet Engineering Task Force (IETF) draft document, available from ftp://ietf.org/internet-drafts/draft-leach-cifs-v1-spec-01.txt.



(a) Startup Latencies



(b) Page Faults

Figure 5: *Effect of Reordering. Page fault data was obtained using the performance counters built into Windows NT. Reordering improves application startup latency and reduces the number of page faults experienced by the program.*

[Lee et al. 98] Lee, D., Crowley, P., Baer, J.-L., Anderson, T., and Bershad, B. Execution Characteristics of Desktop Applications on Windows NT. In *Proc. 25th Annual International Symposium on Computer Architecture*, pages 27–38, June 1998.

[Levy & Redell 82] Levy, H. M. and Redell, D. D. Virtual Memory Management in VAX/VMS. *Computer*, 15(3):35–41, March 1982.

[Melanson 98] Melanson, E. Tuning up. *PC Magazine*, August 1998.

[Microsoft 98] Microsoft. Microsoft Developer Network Library, April 1998. on CD-ROM.

[Nielsen et al. 97] Nielsen, H., Gettys, J., Baird-Smith, A., Prudhommeau, E., Lie, H., and Lilley, C. Network Performance Effects of HTTP/1.1, CSS1, and PNG. In *Proc. of the ACM SIGCOMM 1997 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 155–166, September 1997.

[Peterson & Davie 96] Peterson, L. L. and Davie, B. S. *Computer Networks, A Systems Approach*, chapter 6. Morgan Kaufmann Publishers, Inc., 1996.

[Pettis & Hansen 90] Pettis, K. and Hansen, R. Profile Guided Code Positioning. In *Proc. ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, pages 16–26, 1990.

[Rizzo 99] Rizzo, L. Dummynet: A Simple Approach to the Evaluation of Network Protocols, February 1999. available from http://www.iet.unipi.it/ luigi/ip_dummynet.

[Romer et al. 97] Romer, T., Voelker, G., Lee, D., Wolman, A., Wong, W., Levy, H., and Bershad, B. Instrumentation and Optimization of Win32/Intel Executables using Etch. In *Proc. of the USENIX Windows NT Workshop*, pages 1–7, 1997.

A Web Application Download Times

This graph shows the startup latency from all our experiments with web applications under various network conditions.

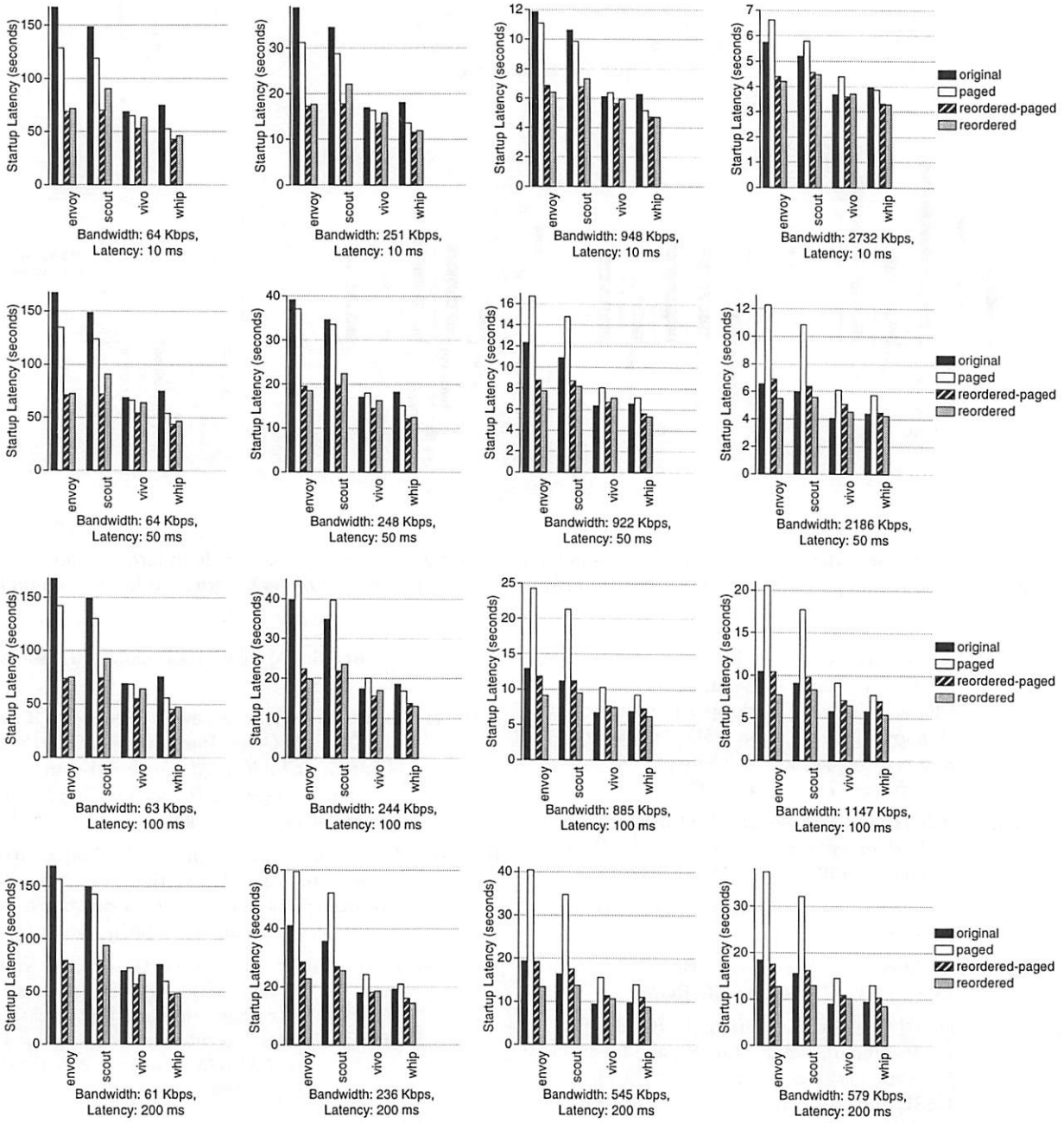


Figure 6: *Web Application Results.*

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

SAGE, the System Administrators Guild

The System Administrators Guild, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must be a member of USENIX.

Member Benefits:

- Free subscription to *;login:*, the Association's magazine, published eight–ten times a year, featuring technical articles, system administration tips and techniques, practical columns on Perl, Java, Tcl/Tk, and C++, book and software reviews, summaries of sessions at USENIX conferences, and Snitch Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, via the USENIX Online Library on the World Wide Web.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as object-oriented technologies, security, operating systems, electronic commerce, and NT – as many as twelve technical meetings every year.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia and other technical publications.
- The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers.
- Discount on BSDI, Inc. products.
- Discount on all publications and software from Prime Time Freeware.
- Savings (10-20%) on selected titles from Academic Press, Morgan Kaufmann, New Riders/Cisco Press/MTP, O'Reilly & Associates, OnWord Press, The Open Group, Sage Science Press, and Wiley Computer Publishing.
- Special subscription rates for Cutter Consortium newsletters, *The Linux Journal, The Perl Journal, IEEE Concurrency, Server/Workstation Expert, Sys Admin Magazine*, and all Sage Science Press journals.

Supporting Members of the USENIX Association:

C++ Users Journal
Cirrus Technologies
Cisco Systems, Inc.
CyberSource Corporation
Deer Run Associates
Greenberg News Networks/MedCast Networks
Hewlett-Packard India Software Operations
Internet Security Systems, Inc.
Microsoft Research
MKS, Inc.
Motorola Australia Software Centre
NeoSoft, Inc.
New Riders Press
Nimrod AS
O'Reilly & Associates Inc.
Performance Computing
Questra Consulting
Sendmail, Inc.
Server/Workstation Expert
TeamQuest Corporation
UUNET Technologies, Inc.
Windows NT Systems Magazine
WITSEC, Inc.

Sage Supporting Members:

Atlantic Systems Group
Collective Technologies
D. E. Shaw & Co.
Deer Run Associates
Electric Lightwave, Inc.
ESM Services, Inc.
GNAC, Inc.
Mentor Graphics Corp.
Microsoft Research
MindSource Software Engineers
Motorola Australia Software Centre
New Riders Press
O'Reilly & Associates Inc.
Remedy Corporation
RIPE NCC
SysAdmin Magazine
Taos Mountain
TransQuest Technologies, Inc.
Unix Guru Universe

For further information about membership, conferences or publications, contact:
USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA.
Phone: 510-528-8649. Fax: 510-548-5738.
Email: *office@usenix.org.*
URL: *http://www.usenix.org.*

ISBN 1-880446-29-4